AD-A205 153

2

AVF Control Number: AVF-VSR-154.0888
88-02-10-HPC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880429W1.09051
Hewlett Packard Company
HP 9000 Series 300 Ada Compiler, Version 3.25
HP 9000 Series 300 Model 360

Completion of On-Site Testing:
04 MAY 88

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Hewlett Packard Company, HP 9000 Series 300 Ada Compiler, Version 3.25, HP 9000 Series 300 Model 360 (Host and Target). (880429W1.09051)		5. TYPE OF REPORT & PERIOD COVERED 4 May 1988 to 4 May 1989
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patteron Air Force Base, Dayton, Ohio, U.S.A.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patteron Air Force Base, Dayton, Ohio, U.S.A.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		12. REPORT DATE 4 May 1988
		13. NUMBER OF PAGES 129 p.
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Wright-Patteron Air Force Base, Dayton, Ohio, U.S.A.		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

HP 9000 Series 300 Ada Compiler, Version 3.25, Hewlett Packard Company, Wright-Patterson Air Force Base, HP 9000 Series 300 Model 360 under HP-UX, Revision 6.2 (Host and Target), ACVC 1.9.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: HP 9000 Series 300 Ada Compiler, Version 3.25

Certificate Number: 880429W1.09051

Host:	Target:
HP 9000 Series 300 Model 360 under HP-UX, Revision 6.2	HP 9000 Series 300 Model 360 under HP-UX, Revision 6.2

Testing Completed 04 MAY 88 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

John F. Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 04 MAY 88 at Hewlett Packard, Cupertino, CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u>.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect

because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: HP 9000 Series 300 Ada Compiler, Version 3.25

ACVC Version: 1.9

Certificate Number: 880429W1.09051

Host Computer:

Machine:	HP 9000 Series 300 Model 360
Operating System:	HP-UX Revision 6.2
Memory Size:	8 Mbytes

Target Computer:

Machine:	HP 9000 Series 300 Model 360
Operating System:	HP-UX Revision 6.2
Memory Size:	8 Mbytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

. Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

. Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64-bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

. Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, LONG_FLOAT, and SHORT_SHORT_INTEGER in the package STANDARD. (See tests B86001C and B86001D.)

. Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

. Expression evaluation.

  Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is

compatible with the target's subtype. (See test C52013A.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

  In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

  All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

  For this implementation:

  An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

  Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

  Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

  Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

  Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

  Length clauses with STORAGE_SIZE specifications for access types are not supported. (See test C87B62B.)

  Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

  Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions, except for the case given in EA3004D, where the function is called inside of a package specification. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests EE2401D and EE2401G.)

There are no strings which are illegal external file names for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102C and CE2102H.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created i˘ OUT_FILE mode, and can be created in IN_FILE mode. (See test E̲3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file can be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 228 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation . Modifications to the code, processing, or grading for 25 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	109	1049	1630	17	16	46	2867
Inapplicable	1	2	223	0	2	0	228
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	499	540	245	166	98	141	327	137	36	234	3	251	2867
Inapplicable	14	73	134	3	0	0	2	0	0	0	0	0	2	228
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	C35904B	C37215E	C45614C	AD1A01A
E28005C	C35A03E	C37215G	A74106C	CE2401H
C34004A	C35A03R	C37215H	C85018B	CE3208A
C35502P	C37213H	C38102C	C87B04B	
A35902C	C37213J	C41402A	CC1311B	
C35904A	C37215C	C45332A	BC3105A	

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 228 tests were inapplicable for the reasons indicated:

- C35702A uses SHORT_FLOAT which is not supported hy this implementation.
- A39005G uses a record representation clause which is not supported by this compiler.

- The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C87B62B uses length clauses with STORAGE_SIZE specifications for access types which are not supported by this implementation.

- EE2401D and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

- The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

TEST INFORMATION

Modifications were required for 24 Class B tests and 1 Class E test.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24007A	B32202A	B37004A	B62001B	B95032A
B24009A	B32202B	B45102A	B74401F	B95069A
B25002A	B32202C	B49003A	B74401R	B95069B
B26005A	B33001A	B49005A	B91004A	BA1101B
B27005A	B36307A	B61012A	B95004A	

In test EA3004D, the pragma has no effect when a function is called inside of a package specification; the error at line 27 in EA3004D6M.ADA is not detected. The test was recompiled in the following order: D0, D1, D4, D5, D2, D3, and D6. The test exhibited the expected behavior, printing NOT_APPLICABLE, and was graded as passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the HP 9000 Series 300 Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the HP 9000 Series 300 Ada Compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a HP 9000 Series 300 Model 360 host operating under HP-UX, Revision 6.2.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tap. were loaded onto an HP Series 9000/320 computer. These files were accessed and referenced across an Ethernet ThinLAN running HP's proprietary Remote File Access (RFA) command. After the test files were loaded to disk, the full set of tests was compiled and linked on the HP 9000 Series 300 Model 360, and all executable tests were run. Results were sent via RFA to an HP 9000/500 Series computer acting as a print server on the network. The print server was used to print all test

output.

The compiler was tested using command scripts provided by Hewlett Packard Company and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

Option	Effect
"-L"	Produce an output listing.
"-P 66"	Set the out page length to 66 lines.
"-e 999"	Set the maximum number of errors to 999.
"-c"	Compile and bind only. The link phase is run separately under the testing automation, using only the default options.

For four tests (AE2101A.ADA, AE2101F.ADA, C35A06N.ADA, and CC1221A.ADA) the option "-u" was used. This option places information for generic instantiations into a separate symbol table. If the option were not used, the test would have caused a symbol table overflow (compiler limitation). Note that all tests could have been run with this option, but doing so would have increased the overall time of the compilations.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Hewlett Packard, Cupertino, CA and was completed on 04 MAY 88.

APPENDIX A

DECLARATION OF CONFORMANCE

Hewlett Packard Company has submitted the following Declaration of Conformance concerning the HP 9000 Series 300 Ada Compiler, Version 3.25.

DECLARATION OF CONFORMANCE

Compiler Implementer:

Hewlett Packard Company
Computer Language Lab

Ada Validation Facility:

ASD/SCEL
Wright-Patterson Air Force Base, OH
45433-6503

Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: HP 9000 Series 300 Ada Compiler, Version 3.25

Host and Target Architectures:

Machine: HP 9000 Series 300 Model 360
Operating System: HP-UX, Revision 6.2

Implementer's Declaration

I, the undersigned, representing Hewlett Packard Company, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compilers listed in this declaration. I declare that Hewlett Packard Company is the owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining said compilers in conformance to ANSI/MIL-STD-1815A. All certificates and registrations of Ada language compilers listed in this declaration shall be made only in the owner's corporate name.

Dave Graham

Hewlett Packard Company
David Graham
Ada R&D Section Manager

Date: 5/6/88

Owner's Declaration

I, the undersigned, representing Hewlett Packard Company, take full responsibility for implementation and maintenance of the Ada compilers listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Dave Graham

Hewlett Packard Company
David Graham
Ada R&D Section Manager

Date: 5/6/88

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the HP 9000 Series 300 Ada Compiler, Version 3.25, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -2_147_483_648 .. 2_147_483_647;
     type SHORT_INTEGER is range -32_768 .. 32_767;
     type SHORT_SHORT_INTEGER is range -128 .. 127;

     type FLOAT is digits 6 range -2#1.111_1111_1111_1111_1111#E+127 ..
                                   2#1.111_1111_1111_1111_1111#E+127;
     type LONG_FLOAT is digits 15 range
-2#1.111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023 ..
 2#1.111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023;

     type DURATION is delta 2#0.000_000_000_000_01# range -86_400.0 .. 86_400.0;

     ...

end STANDARD;
```

Appendix F Implementation-Dependent Characteristics for the HP 9000 Series 300 Compilation System

The Ada Programming Language is described in the *Reference Manual for the Ada Programming Language* (*LRM*). This manual, *Appendix F* of the *LRM*, summarizes the implementation dependencies of the HP Ada Compilation System on the HP 9000 Series 300 Computer System.

Appendix F includes the following information:

- HP implementation-dependent pragmas and attributes
- Specifications of the packages SYSTEM and STANDARD
- Restrictions on representation clauses
- Restrictions on unchecked type conversions
- Naming conventions for implementation-dependent components in record representation clauses
- Implementation-dependent characteristics of input/output packages
- Information about signal handling, the layout and alignment of data types, and the limits imposed by the compiler
- Specific information about calling external subprograms written in HP 68K Assembly Language, HP C, HP FORTRAN 77, and HP Pascal, with examples.

F 1. Implementation-Dependent Pragmas

This section describes the implementation-dependent aspects of pragmas. A *pragma* is a compiler directive that provides a way to control the behavior of one or more components of the Ada compilation system.

The pragmas listed below are described in the sections which follow:

- Pragmas used in interfacing with subprograms written in other languages: pragma INTERFACE and pragma INTERFACE_NAME.
- Pragmas used in conditional compilation: pragma BEGIN_COMPILE, pragma END_COMPILE, pragma NOW_COMPILE, and pragma STOP_COMPILE.
- Pragmas used to support text processing tools: pragma INDENT.
- Pragmas that are not implemented.

F 1.1 Interfacing the Ada Language with Other Languages

Your Ada programs can call subprograms written in other languages when you use the predefined pragmas INTERFACE and INTERFACE_NAME. This implementation of HP Ada supports subprograms written in HP 68K Assembly Language HP C, HP Pascal, and HP FORTRAN 77 for HP 9000 Series 300 computers. Note that compiler products from vendors other than HP may not conform to the parameter passing conventions given below. For detailed information, instructions, and examples for interfacing your Ada programs with HP 68K Assembly Language, HP C, HP FORTRAN 77, and HP Pascal, see Section 14.

The pragma INTERFACE (*LRM*, Section 13.9) informs the compiler that a non-Ada external subprogram unit will be supplied when the Ada program is linked. Pragma INTERFACE specifies two things: the programming language used in the external subprogram and the name of the Ada interfaced subprogram. Implicit in the language specification is the corresponding parameter calling convention to be used in the interface.

The implementation-defined pragma INTERFACE_NAME associates an alternative name with a non-Ada external subprogram that has been specified to the Ada program by the pragma INTERFACE.

The two pragmas take the form:

pragma INTERFACE (*Language_name*, *Ada_subprogram_name*);

pragma INTERFACE_NAME (*Ada_subprogram_name*, *External_name*);

where:

Language_name is one of ASSEMBLER, C, FORTRAN, or PASCAL.

Ada_subprogram_name is the name used within the Ada program when referring to the interfaced external subprogram.

External_name is the external name of the subprogram. This name is literal and case significant.

Use pragma INTERFACE_NAME whenever the interfaced subprogram name contains characters not acceptable within Ada names or when the interfaced subprogram name must contain uppercase letter(s). If you omit pragma INTERFACE_NAME, the name used is the name of the subprogram specified in pragma INTERFACE with all alphabetic characters in the name shifted to lower case and then truncated to 255 characters if necessary. If the interfaced subprogram language is HP C, HP Pascal or HP FORTRAN 77, the compiler modifies the name further: the name is prefixed with one underscore character (_) and then truncated to 255 characters if necessary. This modification conforms to the naming conventions used by the HP linker (Link Editor) on HP-UX systems.

Pragma INTERFACE_NAME is allowed at the same places in an Ada program as pragma INTERFACE (see *LRM* Section 13.9). Pragma INTERFACE_NAME must follow the declaration of the corresponding pragma INTERFACE and must be within the same declarative part, though it need *not* immediately follow that declaration. The following example illustrates the use of these pragmas.

EXAMPLE:

```
package SAMPLE_LIB is

   function SAMPLE_DEVICE  (X : INTEGER) return INTEGER;
   function PROCESS_SAMPLE (X : INTEGER) return INTEGER;

private

   pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE  );
   pragma INTERFACE (C, PROCESS_SAMPLE );

   pragma INTERFACE_NAME (SAMPLE_DEVICE, "Dev10"  );
   pragma INTERFACE_NAME (PROCESS_SAMPLE, "DOsample"  );

end SAMPLE_LIB ;
```

Notice that the external names of the interfaced function subprograms SAMPLE_DEVICE and PROCESS_SAMPLE contain uppercase letters. These external subroutine names differ from the names used within the Ada calling program.

To avoid conflicts with the Ada run-time system, the names of interfaced external subprograms should not begin with the letters "RTS_" or "ALSY" in any combination of uppercase and lowercase.

To avoid conflicts with the HP-UX operating system or the standard HP-UX library of supported subroutines, the names of interfaced subprograms should not be the same as the name of any operating system system call (see Section 2 of the *HP-UX Reference*) or the name of any HP-supplied subroutine library routine (see Section 3 of the *HP-UX Reference*). When you want to call one of these HP-UX library or system routines, direct the binder and linker to utilize code from the system or subroutine library.

For detailed information on the use of pragma INTERFACE and pragma INTERFACE_NAME, see Section F 14.

F 1.2 Pragma INDENT

Pragma INDENT is ignored by the compiler, but it is reserved for support of future Ada source text processing tools.

The following are reserved for use with a source text reformatting tool (often referred to as an Ada pretty printer).

The pragmas INDENT(ON) and INDENT(OFF) affect only the HP supplied reformatter. You can place these pragmas in the source code to control the actions of the reformatter. Table F-1 lists the use of each pragma.

Table F-1. Pragma Indent

Pragma	Description
INDENT(OFF)	The reformatter does not modify the source lines after the pragma.
INDENT(ON)	The reformatter resumes its action after the pragma.

F 1.3 Pragmas Not Implemented

The following pragmas are not implemented and will issue a warning at compile time:

```
CONTROLLED
MEMORY_SIZE
OPTIMIZE
PACK
SHARED
STORAGE_UNIT
SYSTEM_NAME
```

Note that PACKing of record types is done systematically by the HP Ada compiler, but there is no PACKing done for array types.

F 2. Implementation-Dependent Attributes

In addition to the representation attributes discussed in the *LRM*, Sections 13.7.2 and 13.7.3, there are four implementation-defined attributes:

```
'RECORD_SIZE
'VARIANT_INDEX
'ARRAY_DESCRIPTOR
'RECORD_DESCRIPTOR
```

The attributes listed above are discussed in Section F 5, "Conventions for Implementation-Generated Names" because the attributes exist for use in record representation clauses.

Note also that there are limitations on the use of the 'ADDRESS attribute. The attribute 'ADDRESS is implemented for all prefixes that have meaningful addresses. However, the following entities do not have meaningful addresses and will cause a compilation error if used as a prefix to 'ADDRESS:

- A constant that is implemented as an immediate value (that is, the constant does not have space allocated for it).
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.

On the HP 9000 Series 300 implementation, the use of 'ADDRESS on the following units returns a value that has no meaningful interpretation:

- a procedure specification
- a procedure body
- a function specification
- a function body

If you use 'ADDRESS on a program unit, the compiler issues a warning at compile time.

F 3. The SYSTEM and STANDARD Packages

This section contains a complete listing of the two predefined library packages: SYSTEM and STANDARD. These packages both contain implementation-dependent specifications.

F 3.1 The Package SYSTEM

The specification of the predefined library package SYSTEM follows:

```
package SYSTEM is

   -- Standard Ada definitions

   type  NAME      is ( HP9000_300 );
   SYSTEM_NAME:  constant NAME := HP9000_300;

   STORAGE_UNIT: constant := 8;

   MEMORY_SIZE:  constant := (2**31)-1;

   MIN_INT:      constant := -(2**31);

   MAX_INT:      constant :=  2**31-1;

   MAX_DIGITS:   constant := 15;

   MAX_MANTISSA: constant := 31;

   FINE_DELTA:   constant := 2#1.0#e-31;

   TICK: constant := 0.020;     -- 20 milliseconds

   subtype PRIORITY is INTEGER range 1..127 ;

   type ADDRESS is private;

   NULL_ADDRESS : constant ADDRESS; --set to NULL

   -- Address arithmetic

   function TO_INTEGER (LEFT : ADDRESS) return INTEGER;

   function TO_ADDRESS (LEFT : INTEGER) return ADDRESS;

   --  Note that  ADDRESS + ADDRESS is not supported
   function "+"   (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
   function "+"   (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

   --  Note that  INTEGER - ADDRESS is not supported
   function "-"   (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
   function "-"   (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
```

```
    function "<"   (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
    function "<="  (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
    function ">"   (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
    function ">="  (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;

    function "mod" (LEFT : ADDRESS; RIGHT : POSITIVE) return NATURAL;

    function IS_NULL (LEFT : ADDRESS) return BOOLEAN;

    function WORD_ALIGNED (LEFT : ADDRESS) return BOOLEAN;

    function ROUND (LEFT : ADDRESS) return ADDRESS;
    -- return the given address rounded to the next lower even value

    procedure COPY (FROM : ADDRESS; TO : ADDRESS; SIZE : NATURAL);
    -- copy SIZE storage units. The result is undefined if
    -- the two storage areas overlap.

    -- functions to provide some READ/WRITE operations in memory.

    generic
       type ELEMENT_TYPE is private;
    function FETCH (FROM : ADDRESS) return ELEMENT_TYPE;
    -- Return the bit pattern stored at address FROM, as a value of
    -- specified ELEMENT_TYPE. This function is NOT implemented for
    -- unconstrained array types.

    generic
       type ELEMENT_TYPE is private;
    procedure STORE (INTO : ADDRESS; OBJECT : ELEMENT_TYPE);
    -- Store the bit pattern representing the value of OBJECT,at
    -- address INTO. This function is NOT implemented for
    -- unconstrained array types.

private

    -- private part of package SYSTEM
    ...

end SYSTEM;
```

F 3.2 The Package STANDARD

The specification of the predefined library package STANDARD follows:

```
package STANDARD is

   -- The operators that are predefined for the types declared in this
   -- package are given in comments since they are implicitly declared.
   -- Lower case is used for pseudo-names of anonymous types (such as
   -- universal_real, universal_integer, and universal_fixed) and for
   -- undefined information (such as any_fixed_point_type).


   -- Predefined type BOOLEAN
   -- Warning: internal representation of BOOLEAN is NOT simply 0,1
   type BOOLEAN is (FALSE, TRUE);

   -- The predefined relation operators for this type are as follows
   -- (these are implicitly declared):
   -- function "="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "<"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function ">"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

   -- The predefined logical operands and the predefined logical
   -- negation operator are as follows (these are implicitly
   -- declared):

   -- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

   -- function "not" (RIGHT : BOOLEAN) return BOOLEAN;



   -- Predefined universal types


   -- type universal_integer is predefined;

   -- The predefined operators for the type universal_integer are as follows
   -- (these are implicitly declared):
   -- function "="  (LEFT, RIGHT : universal_integer) return BOOLEAN;
   -- function "/=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
   -- function "<"  (LEFT, RIGHT : universal_integer) return BOOLEAN;
   -- function "<=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
   -- function ">"  (LEFT, RIGHT : universal_integer) return BOOLEAN;
   -- function ">=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
```

```
-- function "+"   (RIGHT : universal_integer) return universal_integer;
-- function "-"   (RIGHT : universal_integer) return universal_integer;
-- function "abs" (RIGHT : universal_integer) return universal_integer;

-- function "+"   (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "-"   (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "*"   (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "/"   (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "rem" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "mod" (LEFT, RIGHT : universal_integer) return universal_integer;

-- function "**"  (LEFT  : universal_integer;
--                 RIGHT : INTEGER) return universal_integer;


-- type universal_real is predefined;

-- The predefined operators for the type universal_real are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : universal_real) return BOOLEAN;

-- function "+"   (RIGHT : universal_real) return universal_real;
-- function "-"   (RIGHT : universal_real) return universal_real;
-- function "abs" (RIGHT : universal_real) return universal_real;

-- function "+"   (LEFT, RIGHT : universal_real) return universal_real;
-- function "-"   (LEFT, RIGHT : universal_real) return universal_real;
-- function "*"   (LEFT, RIGHT : universal_real) return universal_real;
-- function "/"   (LEFT, RIGHT : universal_real) return universal_real;

-- function "**"  (LEFT : universal_real;
--                 RIGHT : INTEGER) return universal_real;

-- In addition, the following operators are predefined for universal types:

-- function "*"   (LEFT  : universal_integer;
--                 RIGHT : universal_real) return universal_real;
-- function "*"   (LEFT  : universal_real;
--                 RIGHT : universal_integer) return universal_real;
-- function "/"   (LEFT  : universal_real;
--                 RIGHT : universal_integer) return universal_real;


-- type universal_fixed is predefined;

-- The only operators declared for this type are:
-- function "*"   (LEFT  : any_fixed_point_type;
--                 RIGHT : any_fixed_point_type) return universal_fixed;
```

```
-- function "/"   (LEFT  : any_fixed_point_type;
--                  RIGHT : any_fixed_point_type) return universal_fixed;


-- Predefined and additional integer types

type SHORT_SHORT_INTEGER is range   -128 .. 127; -- 8 bits long
-- this is equivalent to -(2**7) .. (2**7 -1)
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "-"   (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "abs" (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;

-- function "+"   (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;
-- function "-"   (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;
-- function "*"   (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;
-- function "/"   (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;
-- function "rem" (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;
-- function "mod" (LEFT, RIGHT : SHORT_SHORT_INTEGER)
--                  return SHORT_SHORT_INTEGER;

-- function "**"  (LEFT  : SHORT_SHORT_INTEGER;
--                 RIGHT : INTEGER) return SHORT_SHORT_INTEGER;


type SHORT_INTEGER       is range-32_768 ..  32_767; --16 bits long

-- this is equivalent to -(2**15) .. (2**15 -1)
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "-"   (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "abs" (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
```

```
-- function "+"   (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "-"   (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "*"   (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "/"   (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "rem" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "mod" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;

-- function "**"  (LEFT  : SHORT_INTEGER;
--                 RIGHT : INTEGER) return SHORT_INTEGER;


type INTEGER    is range -2_147_483_648 ..  2_147_483_647; --32 bits long

-- this is equivalent to -(2**31) .. (2**31 -1)
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : INTEGER) return INTEGER;
-- function "-"   (RIGHT : INTEGER) return INTEGER;
-- function "abs" (RIGHT : INTEGER) return INTEGER;

-- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "**"  (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;


-- Predefined INTEGER subtypes
subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;



-- Predefined and additional floating point types

type FLOAT is digits 6 range -- 32 bits long
-2#1.111_1111_1111_1111_1111_1111#E+127 ..
 2#1.111_1111_1111_1111_1111_1111#E+127;
--
-- This is equivalent to -(2.0 - 2.0**(-23)) * 2.0**127 ..
--       +(2.0 - 2.0**(-23)) * 2.0**127 ..
--
--  This is approximately equal to the decimal range:
--  -3.402823E+38 .. +3.402823E+38
```

```
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;

-- function "+"   (RIGHT : FLOAT) return FLOAT;
-- function "-"   (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

-- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;

-- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;


type LONG_FLOAT is digits 15 range -- 64 bits long
-2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023
 ..
 2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023;
--
-- This is equivalent to -(2.0 - 2.0**(-52)) * 2.0**1023 ..
--       +(2.0 - 2.0**(-52)) * 2.0**1023 ..
-- This is approximately equal to the decimal range:
-- -1.797693134862315E+308 .. +1.797693134862315E+308

-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "="  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;

-- function "+"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "abs" (RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "+"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "*"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "/"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "**"  (LEFT : LONG_FLOAT; RIGHT : INTEGER) return LONG_FLOAT;


--This implementation does not provide any other floating point types
```

```
-- Predefined type DURATION
type DURATION is delta 2#0.000_000_000_000_01# range -86_400.0 .. 86_400.0;
--
-- DURATION'SMALL derived from this delta is  2.0**(-14) , which is the
-- maximum precision that an object of type DURATION can have and still
-- be representable in this implementation. This has an approximate
-- decimal equivalent of  0.000061 (61 microseconds).
-- The predefined operators for the type DURATION are the same as for any
-- fixed point type.

-- This implementation provides many anonymous predefined fixed point
-- types.  They consist of all types whose "small" value is
-- any power of 2.0 between 2.0 ** (-31) and 2.0 ** (31),
-- and whose mantissa can be expressed using 31 or less binary digits.

-- The following lists characters for the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers; they are indicated in lower case in this section.


-- Predefined type CHARACTER
type CHARACTER is
  ( nul, soh, stx, etx,  eot, enq, ack, bel,
    bs,  ht,  lf,  vt,   ff,  cr,  so,  si,
    dle, dc1, dc2, dc3,  dc4, nak, syn, etb,
    can, em,  sub, esc,  fs,  gs,  rs,  us,

    ' ', '!', '"', '#',  '$', '%', '&', ''',
    '(', ')', '*', '+',  ',', '-', '.', '/',
    '0', '1', '2', '3',  '4', '5', '6', '7',
    '8', '9', ':', ';',  '<', '=', '>', '?',

    '@', 'A', 'B', 'C',  'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K',  'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S',  'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[',  '\', ']', '^', '_',

    '`', 'a', 'b', 'c',  'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k',  'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's',  't', 'u', 'v', 'w',
    'x', 'y', 'z', '{',  '|', '}', '~', DEL);

--The predefined operators for the type CHARACTER are the same as
--for any enumeration type.


-- Predefined type STRING (RM 3.6.3)
type STRING is array (POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:
-- function "="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
```

```
-- function ">"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;

-- Predefined catenation operators
-- function "&" (LEFT : STRING;    RIGHT : STRING) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : STRING) return STRING;
-- function "&" (LEFT : STRING;    RIGHT : CHARACTER) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;


-- Predefined exceptions
CONSTRAINT_ERROR : exception;
NUMERIC_ERROR    : exception;
PROGRAM_ERROR    : exception;
STORAGE_ERROR    : exception;
TASKING_ERROR    : exception;


-- Predefined package ASCII
package ASCII is

   -- Control characters
   NUL  : constant CHARACTER := nul;
   SOH  : constant CHARACTER := soh;
   STX  : constant CHARACTER := stx;
   ETX  : constant CHARACTER := etx;
   EOT  : constant CHARACTER := eot;
   ENQ  : constant CHARACTER := enq;
   ACK  : constant CHARACTER := ack;
   BEL  : constant CHARACTER := bel;
   BS   : constant CHARACTER := bs;
   HT   : constant CHARACTER := ht;
   LF   : constant CHARACTER := lf;
   VT   : constant CHARACTER := vt;
   FF   : constant CHARACTER := ff;
   CR   : constant CHARACTER := cr;
   SO   : constant CHARACTER := so;
   SI   : constant CHARACTER := si;
   DLE  : constant CHARACTER := dle;
   DC1  : constant CHARACTER := dc1;
   DC2  : constant CHARACTER := dc2;
   DC3  : constant CHARACTER := dc3;
   DC4  : constant CHARACTER := dc4;
   NAK  : constant CHARACTER := nak;
   SYN  : constant CHARACTER := syn;
   ETB  : constant CHARACTER := etb;
   CAN  : constant CHARACTER := can;
   EM   : constant CHARACTER := em;
   SUB  : constant CHARACTER := sub;
   ESC  : constant CHARACTER := esc;
   FS   : constant CHARACTER := fs;
   GS   : constant CHARACTER := gs;
   RS   : constant CHARACTER := rs;
```

```
    US    : constant CHARACTER := us;
    DEL   : constant CHARACTER := del;

    -- other characters
    EXCLAM        : constant CHARACTER :='!';
    QUOTATION     : constant CHARACTER :='"';
    SHARP         : constant CHARACTER :='#';
    DOLLAR        : constant CHARACTER :='$';
    PERCENT       : constant CHARACTER :='%';
    AMPERSAND     : constant CHARACTER :='&';
    COLON         : constant CHARACTER :=':';
    SEMICOLON     : constant CHARACTER :=';';
    QUERY         : constant CHARACTER :='?';
    AT_SIGN       : constant CHARACTER :='@';
    L_BRACKET     : constant CHARACTER :='[';
    BACK_SLASH    : constant CHARACTER :='\';
    R_BRACKET     : constant CHARACTER :=']';
    CIRCUMFLEX    : constant CHARACTER :='^';
    UNDERLINE     : constant CHARACTER :='_';
    GRAVE         : constant CHARACTER :='`';
    L_BRACE       : constant CHARACTER :='{';
    BAR           : constant CHARACTER :='|';
    R_BRACE       : constant CHARACTER :='}';
    TILDE         : constant CHARACTER :='~';

    -- Lower case letters
    LC_A : constant CHARACTER := 'a';
    LC_B : constant CHARACTER := 'b';
    LC_C : constant CHARACTER := 'c';
    LC_D : constant CHARACTER := 'd';
    LC_E : constant CHARACTER := 'e';
    LC_F : constant CHARACTER := 'f';
    LC_G : constant CHARACTER := 'g';
    LC_H : constant CHARACTER := 'h';
    LC_I : constant CHARACTER := 'i';
    LC_J : constant CHARACTER := 'j';
    LC_K : constant CHARACTER := 'k';
    LC_L : constant CHARACTER := 'l';
    LC_M : constant CHARACTER := 'm';
    LC_N : constant CHARACTER := 'n';
    LC_O : constant CHARACTER := 'o';
    LC_P : constant CHARACTER := 'p';
    LC_Q : constant CHARACTER := 'q';
    LC_R : constant CHARACTER := 'r';
    LC_S : constant CHARACTER := 's';
    LC_T : constant CHARACTER := 't';
    LC_U : constant CHARACTER := 'u';
    LC_V : constant CHARACTER := 'v';
    LC_W : constant CHARACTER := 'w';
    LC_X : constant CHARACTER := 'x';
    LC_Y : constant CHARACTER := 'y';
    LC_Z : constant CHARACTER := 'z';

end ASCII;
```

```
   --Certain aspects of the predefined entities cannot be completely
   --described in the language itself.  For example, although the
   --enumeration type BOOLEAN can be written showing the two
   --enumeration literals FALSE and TRUE, the short-circuit control
   --forms cannot be expressed in the language.

end STANDARD;
```

F 4. Restrictions on Representation Clauses

The facilities for supporting representation clauses are covered in the *LRM*, Chapter 13. The exceptions are listed below:

- There is no bit level implementation for any representation clause in this implementation.
- The alignment clause in the record representation clause can only be used to align records on one-byte or two-byte boundaries; this value must be specified in storage units.
- Address clauses are not supported.
- Change in representation for record types is not implemented.
- Machine code insertions are not implemented.
- For the length clause:
  - Size specification: T'SIZE is not implemented for types declared in a generic unit.
  - Specification of storage for a task activation: T'STORAGE_SIZE is not implemented when T is a task type. If 'STORAGE_SIZE is used on an access type without setting a STORAGE_SIZE, a value of zero is returned.
  - Specification of SMALL for a fixed point type: T'SMALL is restricted to a power of 2, and the absolute value of the exponent must be less than 31.
- The enumeration representation clause is not allowed if there is a range constraint on the parent subtype.
- The record representation clause is not allowed for a derived record type.

F 5. Conventions for Implementation-Generated Names

The names described in the following sections are reserved for use in this implementation of Ada and, in general, should not be used in your programs.

F 5.1 Names for Record Components and Attributes

The compiler introduces special implicit record components for certain record type definitions. These record components are implementation-dependent; they are used by the compiler to improve the quality of the code generated for certain operations on the record types. The existence of these components depends on implementation-dependent criteria. Attributes have been defined for these fields for use in record representation clauses.

If you refer to an implementation-dependent attribute that does not exist, the compiler issues an error message. If the implementation-dependent component exists, the compiler checks that the storage space specified in the component clause is compatible with the treatment of this component and that the storage space of the other components is compatible. The compiler issues an error message if this check fails.

Table F-2 lists the four implementation-defined attributes. These attributes exist only when the conditions listed in the description are satisfied.

Table F-2. Attributes and Record Types

Attribute	Description
T'RECORD_SIZE	Prefix T denotes a record type. A record component is introduced by the compiler in a record to store the size of the record object. This component exists for objects with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.
T'VARIANT_INDEX	Prefix T denotes a record type. A record component is introduced by the compiler in a record for the efficient implementation of discriminant checks. This component exists for objects of a record type with variant parts.
C'ARRAY_DESCRIPTOR	Prefix C denotes a record component of array type whose component subtype definition depends on discriminants. This record component is introduced by tne compiler in a record to store information on subtypes of components that depend on discriminants.
C'RECORD_DESCRIPTOR	Prefix C denotes a record component of record type whose component subtype definition depends on discriminants. This record component is introduced by the compiler in a record to store information on subtypes of components that depend on discriminants.

Table F-3 lists the four implementation-generated names. These names are reserved by the HP Ada/300 implementation and are not available for user-created components in records. However, these names are available as attributes when you use representation clauses. The names are visible for records only when they are appropriate to that record type.

Table F-3. Implementation-Generated Names

Names	Description
RECORD_SIZE	An implementation-specific record component. This component is introduced by the compiler in a record to store the size of the record object.
VARIANT_INDEX	An implementation-specific record component. This component is introduced by the compiler in a record for the efficient implementation of discriminant checks.
ARRAY_DESCRIPTOR	An internal component used by the compiler to store information on subtypes of record components that depend upon discriminants. Used for record components of array types.
RECORD_DESCRIPTOR	An internal component used by the compiler to store information on subtypes of record components that depend upon discriminants. Used for record components of record types.

F 5.2 Names for Predefined Library Units

The following names are used by the HP Ada Compilation System. Do not use any of these names for your library-level Ada units.

```
ALSYS_ADA_RUNTIME
ALSYS_BASIC_IO
ALSYS_BINARY_IO
ALSYS_COMMON_IO
ALSYS_FILE_MANAGEMENT
ALSYS_IO_TRACES
HIT
MATH_EXCEPTIONS *
MATH_LIB *
UNIX_ENV *
```

The packages whose names are followed by * are available to be used in your programs. These packages are documented in the *Ada User's Guide*.

F 6. Interpretation of Expressions Appearing in Address Clauses

Address clauses are not implemented in this version of the HP Ada compiler. Refer to the *LRM*, Section 13.5 for information on address clauses.

F 7. Restrictions on Unchecked Type Conversions

Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are scalar or access types, the source objects must be the same size as the target objects.

If a composite type is used either as source or as target type, the restriction on the size does not apply.

If the source and the target types are each scalar or access types or if they are both composite types, the effect of the function is to return the operand.

In other cases, the effect of unchecked conversion can be considered as a copy.

WARNING

When you do an UNCHECKED_CONVERSION among types whose sizes do not match, the code which is generated copies as many byes as necessary from the source location to fill the target. If the target is larger than the source, the code copies all of the source plus whatever happens to follow the source. So an UNCHECKED_CONVERSION among types whose sizes do not match can produce meaningless results, or actually cause a trap and abort the program (if these memory locations do not actually exist).

F 8. Implementation-Dependent Input-Output Characteristics

Section 8 covers the input/output (I/O) characteristics of Ada on the HP 9000 Series 300 computer. Ada handles I/O with packages, which are discussed in Section F 8.1. File types are covered in Section F 8.1.3. The FORM parameter is discussed in detail in Section F 8.2.

F 8.1 Ada I/O Packages for External Files

I/O operations are considered to be performed on objects of a specific file type, rather than being performed directly on external files. An external file is a file external to the program that can produce a value to be read or receive a value to be written. Values transferred for a given file must be all of one type.

Generally in Ada documentation, the term file refers to an object of a certain file type, whereas a physical manifestation is known as an external file. An external file is characterized by:

- Its NAME, which is a string defining a legal pathname under the current version of the operating system. For example, the external file name `myfile` may have the actual rooted path /PROJECT/`myfile`. In that case, use of `myfile` as a string specifying an external file would be legal if the current working directory is /PROJECT. See the example in Section F 8.1.2.

- Its FORM, which gives implementation-dependent information on file characteristics.

Both NAME and FORM appear explicitly in the Ada CREATE and OPEN procedures. Though a file is an object of a certain file type, ultimately the object has to correspond to an external file. Both CREATE and OPEN associate the NAME of an external file (of a certain FORM) with a program file object.

Ada I/O operations are provided by standard packages. (See the *LRM*, Chapter 14 for more details.) Table F-4 describes the standard Ada I/O packages.

Table F-4. Standard I/O Packages

Package	Description and LRM Location
SEQUENTIAL_IO	A generic package for sequential files of a single element type. (Section 14.2.3)
DIRECT_IO	A generic package for direct (random) access files of a single element type. (Section 14.2.5)
TEXT_IO	A non-generic package for ASCII text files. (Section 14.3.10)
IO_EXCEPTIONS	A package which defines the exceptions needed by the above three packages. (Section 14.5)

The generic package LOW_LEVEL_IO is not implemented.

F 8.1.1 Implementation-Dependent Restrictions on I/O Packages

The upper bound for index values in DIRECT_IO and for line, column, and page numbers in TEXT_IO is:

```
COUNT'LAST = 2**31 -1
```

The upper bound for fields' widths in TEXT_IO is:

```
FIELD'LAST = 255
```

F 8.1.2 Correspondence between External Files and HP-UX Files

Ada I/O is considered in terms of external files. Data is read from and written to external files. Each external file is implemented as a standard HP-UX file. However, before an external file can be used by an Ada program, it must be associated with a file object belonging to that program. This association is achieved by supplying the name of the file object and the name of the external file to the procedures CREATE or OPEN of the predefined I/O packages. Once the association has been made, the external file can be read from or written to with the file object. Note that for SEQUENTIAL_IO and DIRECT_IO you must first instantiate the generic package to produce a non-generic instance. Then you can use the CREATE or OPEN procedure of that instance. The example at the end of this section illustrates this instantiation process.

The name of the external file can be either of the following:

- Null string (CREATE only)
- HP-UX pathname

If the name is a null string, the associated external file is a temporary file, created using the HP-UX facility tmpnam(3). This external file will cease to exist upon completion of the program.

If the external file is an HP-UX pathname, the pathname is extended in conformance with HP-UX rules (see intro (2) in the *HP-UX Reference*). The exception USE_ERROR is raised by the procedure CREATE if the specified external file is a device. USE_ERROR is also raised for either OPEN or CREATE if you have insufficient access rights to the file.

If an *existing external file* is specified to the CREATE procedure, the contents of that file will be *deleted*. The recreated file is left open as for a newly created file, for later access by the program that made the call to create the file.

NOTE

> Executing the procedures and functions of the predefined I/O packages involves the use of the Ada run-time system and the possible execution of a number of HP-UX I/O primitives. When such primitives are executed, an HP-UX I/O signal can be raised (such signals are raised, for example, if errors are detected during the processing of an I/O primitive).
>
> If certain HP-UX I/O signals are raised, they are caught and processed by the Ada run-time system. This processing causes the appropriate Ada exception to be raised. Such Ada exceptions can then be handled by the Ada program.

EXAMPLE

```
-- Example of creating a file using the generic package DIRECT_IO
-- The example also illustrates file close and reopening in a different
-- access mode
with DIRECT_IO;
with TEXT_IO;
procedure RTEST is

    package INTIO is new DIRECT_IO (INTEGER);  --instantiate on INTEGER
    use INTIO;                                 --types

    IFILE : INTIO.FILE_TYPE;  -- Define a file object for reference in Ada
    IVALUE : INTEGER := 0;    -- Create an integer element

begin  -- RTEST

    CREATE ( FILE => IFILE,             -- Refer to file object IFILE
             MODE => INTIO.INOUT_FILE, -- Set read/write access mode
             NAME => "myfile"          -- Associate an external file name
           );                          -- "myfile" to the file object, IFILE
    TEXT_IO.PUT_LINE ("Created :" & INTIO.NAME (IFILE));

    CLOSE ( FILE => IFILE);            -- Close the external file
    TEXT_IO.PUT_LINE("Closed file");

    OPEN  (FILE => IFILE,              -- Open the file object
           MODE => INTIO.OUT_FILE,     -- in the new MODE to allow write only
           NAME => "myfile"            -- Associate an external file name
          );                           -- "myfile"  to the file object, IFILE
     TEXT_IO.PUT_LINE("Opened (new mode) :" & INTIO.NAME (IFILE));

     INTIO.WRITE (IFILE, IVALUE) ;  -- Write an integer to the file
     TEXT_IO.PUT_LINE("Appended an Integer to:" & INTIO.NAME (IFILE));

     CLOSE ( FILE => IFILE);
     TEXT_IO.PUT_LINE("Close file");

end RTEST;
```

In the example above, the file object is IFILE, the external file name relative to your current working directory is myfile, and the actual rooted path could be /PROJECT/myfile. Error or informational messages from the Ada development system (compiler, tools) may mention the actual rooted path.

NOTE

The Ada/300 development system manages files internally so that names involving symbolic links (see *ln(1)*) are mapped back to the actual rooted path. Consequently, when the development system interacts with files involving symbolic links, the actual rooted pathname may be mentioned in informational or error messages rather than the symbolic name.

F 8.1.3 Standard Implementation of External Files

External files have a number of implementation-dependent characteristics, such as their physical organization and file access rights. It is possible to customize these characteristics through the FORM parameter of the CREATE and OPEN procedures, described fully in Section F 8.2. The default of FORM is the null string. The following section describes the standard or default implementation of three types of external files: sequential, direct, and text. Default protection for external files is also described.

NOTE

In the absence of the FORM parameter, default protection is a function of the HP-UX operating system (see Section F 8.2 which follows).

F 8.1.3.1 Sequential Files

A sequential file is a sequence of values that are transferred in the order of their appearance (as produced by the program or by the run-time environment). A file is a collection of data elements (object components) of identical type. Each object in a sequential file has exactly the same binary representation as the Ada object in the executable program.

The information placed in a sequential file depends on whether the type used for the instantiation of the sequential I/O package is constrained or unconstrained. If it is constrained, the objects are put consecutively into the file, without holes or separators. If the type is unconstrained, the length of the object (in bytes) is added to the front of the object as a 32-bit integer value. See Section F 8.2.5 for a detailed description. The default is that there is no buffer between the physical external file and the Ada program. However, see Section F 8.2.6 for details on specifying a file BUFFER_SIZE.

F 8.1.3.2 Direct Files

A direct access file is a set of elements occupying consecutive positions in a linear order. The position of an element in a direct file is specified by its index, which is an integer in the range 1 to (2**31)-1 of subtype POSITIVE_COUNT. If the file is created with the default FORM parameter attributes (see Section F 8.2), only objects of a constrained type can be written to or read from a direct access file. Such objects have exactly the same binary representation as the Ada object in the executable program. Although instantiation of DIRECT_IO is accepted for unconstrained types, the exception USE_ERROR is raised on any call to CREATE or OPEN where the object is of an unconstrained type. Using the FORM parameter allows you to store unconstrained objects in a direct access file by specifying the maximum RECORD_SIZE of unconstrained types.

A file is a collection of data elements (object components) of identical type. All elements within the file have the same length. The number of bytes occupied by each element is determined by the size of the object stored in the file. The default is that there is no buffer between the physical external file and the Ada program. See Section F 8.2.6 for details on specifying a file BUFFER_SIZE.

F 8.1.3.3 Text Files

Text files are used for the input and output of information which is in a readable form. Each text file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. All text file column numbers, line numbers, and page numbers are in the range 1 to (2**31)-1 of subtype POSITIVE_COUNT. The line terminator (end-of-line) is physically represented by the ASCII character ASCII.LF. The page terminator (end-of-page) is physically represented by a succession of the two characters, ASCII.LF and ASCII.FF, in that order. The file terminator (end-of-file) is physically represented by the character ASCII.LF, followed by the HP-UX END-OF-FILE. See Section F 8.2.5 in this appendix for more information about structuring text files.

If you control line, page, and file structure by calling predefined subprograms (*LRM*, Section 14.3.4), you need not be concerned with the above terminator implementation details. If you effect structural control by explicitly inputting or outputting these control characters (via the PUT function, for example), it is your responsibility to maintain the integrity of the external file. The standard implementation of text files does not buffer text file I/O. However, buffering is a device-dependent characteristic that can be modified at the system level (for example, to buffer lines of text entered from a terminal).

CAUTION

If a terminal is used for text input, the functions END_OF_PAGE and END_OF_FILE always return FALSE.

F 8.1.4 Default Access Protection of External Files

HP-UX provides protection of a file by means of *access rights*. These access rights are used within Ada programs to protect external files. There are three levels of protection:

- User (the owner of the file).
- Group (users belonging to the owner's group).
- Others (users belonging to other groups).

For each of these levels, access to the file can be limited to one or several of the following rights: read, write, or execute. The standard external file access rights are specified by the UMASK command (see umask(1) and umask(2) in the *HP-UX Reference*). Access rights apply equally to sequential, direct, and text files. See the section on the FORM parameter (F 8.2) for information about specifying file permissions at the time of CREATE.

F 8.1.5 The Sharing of External Files and Tasking Issues

Several file objects can be associated with the same external file. The objects can have identical or differing I/O modes; each file object has independent access to the external file. The effects of sharing an external file depend on the nature of the file. You must keep in mind the nature of the device attached to the file object and the sequence of I/O operations on the device. Multiple I/O operations on an external file shared by several file objects are processed in the order they occur.

In the case of shared files on random access devices, such as discs, the data is shared. Reading from one file object does not affect the file positioning of another file object, nor the data available to it. However, simultaneous reading and writing of separate file objects to the same random access external file (a direct

access file, for example) should be avoided; due to buffering, the effects are unpredictable. File buffering may be enabled by using the FORM parameter attributes at the time you open or create the file.

In the case of shared files as sequential or interactive devices, such as magnetic tapes or keyboards, the data is no longer shared. In other words, a magnetic record or keyboard input buffer read by one I/O operation is no longer available to the next operation, whether it is performed on the same file object or not. This is simply due to the sequential nature of the device. By default, file objects represented by STANDARD_IN and STANDARD_OUT are preconnected to the HP-UX streams stdin and stdout (see stdio(5)), and thus are of this sequential variety of file. The HP-UX stream stderr is not preconnected to an Ada file but is used by the Ada run-time system for error messages.

Each file operation is completed before a subsequent file operation commences. In a tasking program, this means no explicit synchronization needs to be performed unless the order of I/O operations is critical. An Ada tasking program is erroneous if it depends on the order of I/O operations without explicitly synchronizing them. Remember that the files associated with STANDARD_IN and STANDARD_OUT are shared by all tasks, and that care must be taken when Ada file objects use buffered I/O.

NOTE

The sharing of external files discussed here is system-wide and is managed by the HP-UX operating system. Several programs may share one or more external files. The file sharing using the FORM parameter SHARED, which is discussed in F 8.2.4, is not system-wide, but is a file sharing within an Ada program and is managed by that program. Synchronized access to a file shared using the FORM parameter SHARED is your responsibility.

F 8.1.6 I/O Involving Access Types

When an object of an access type is specified as the source or destination of an I/O operation (read or write), the 32-bit binary access value is read or written unchanged. If an access value is read from a file, take care to ensure that the access value so read designates a valid object. This is only likely to be the case if the access value read was previously written by the same program that is reading it, and the object which it designated at the time it was written still exists (that is, the scope in which it was allocated has not been exited, nor has an UNCHECKED_DEALLOCATION been performed on it). A program may execute erroneously if an access type read from a file does not designate a valid object.

F 8.1.7 I/O Involving Local Area Networks

This section assumes knowledge of both remote file access and networks. It describes Ada program I/O involving two Local Area Network (LAN) services available on the Series 300 computers:

1. RFA systems: remote file access (RFA) using the NS/9000 network services software.

2. NFS® systems: remote file access using the NFS network services software.

NFS® is a trademark of SUN Microsystems, Inc.

The Ada programs discussed here are executed on a local (host) computer. These programs access or create files on a remote system, which is located on a mass storage device not directly connected to the host computer. The remote file system can be mounted and accessed by the host computer using RFA or NFS LAN services. RFA systems are described in *Network Services/9000 LAN User's Guide for HP 9000 Computers*. NFS systems are described in *Using and Administering NFS Services*.

Note that Ada I/O can be used reliably across local area networks using RFA only if the network special files representing remote file systems are contained in the directory /net, as is customary on HP-UX systems. See the *HP-UX System Administrator Manual* (for the Series 300) for more details.

F 8.1.7.1 RFA Systems

Properly specified external files can be created or accessed reliably from Ada programs across the LAN on RFA systems. You can create or access a file only if an RFA connection exists from the remote file system at the time your Ada program is executed. The example on the following page illustrates remote file access and use.

In this example, a remote network connection to a system hpclada is assumed. This connection could have been made by typing, for example, (where $ is the shell prompt),

```
$ netunam /net/hpclada user_name:
Password: user_passwd
$
```

EXAMPLE

```
with DIRECT_IO;
procedure LANTEST is

   -- instantiate the generic package DIRECT_IO
   -- for files whose component type is INTEGER.
   package TESTIO is new DIRECT_IO(INTEGER);
   use TESTIO;

   REMOTE_FILE : TESTIO.FILE_TYPE;

   IVALUE : INTEGER :=  0 ;
   RVALUE : INTEGER := -1 ;

begin

    -- create a remote file
    TESTIO.CREATE (FILE => REMOTE_FILE,
                   MODE => TESTIO.OUT_FILE, --OUT mode for DIRECT_IO
                   NAME => "/net/hpclada/project/test.file");

    -- Close the file
    TESTIO.CLOSE  (FILE => REMOTE_FILE);

    -- Re-open the file with different file mode
    TESTIO.OPEN   (FILE => REMOTE_FILE,
                   MODE => TESTIO.INOUT_FILE, --INOUT mode for DIRECT_IO
                   NAME => "/net/hpclada/project/test.file");
```

```
        -- Write an integer (0) to the file
        TESTIO.WRITE (REMOTE_FILE, IVALUE);

        -- Reset the file pointer in the file
        TESTIO.RESET (REMOTE_FILE);

        -- Read from the file, rvalue should now be zero
        TESTIO.READ (REMOTE_FILE, RVALUE);

        -- Close the file
        CLOSE  (FILE => REMOTE_FILE);

    end LANTEST;
```

F 8.1.7.2 NFS Systems

If an Ada program expects to access or create a file on a remote file system using NFS LAN services, the remote volumes which contain the file system must be mounted on the host computer prior to the execution of the Ada program.

For example, assume that the remote system (hpclada) exports a file system /project. /project is mounted on the host computer as /ada/project. Files in this remote file system are accessed or created by references to the files as if they were part of the local file system. To access the file test.file, the program would reference /ada/project/test.file on the local system. Note that test.file appears as /project/test.file on the remote system. The *netunam(1)* command (from HP's NS/9000) is not used in NFS.

The remote file system must be exported to the local system before it can be locally mounted, using the *mount(1m)* command.

F 8.1.8 Implementation-Defined I/O Packages

UNIX_ENV is the only implementation-defined I/O package. An Ada program that does not use UNIX_ENV has no implicit access to its execution environment without using the package UNIX_ENV. In particular, there is no direct equivalent to the argc and argv parameters of a C Language main program. In this implementation, parameters cannot be passed to a procedure used as an Ada main program unless the HP-supplied package UNIX_ENV is used.. The binder will not bind a program with parameters in its main program. The implementation-defined package UNIX_ENV enables an Ada program to retrieve information from the HP-UX environment. The specification of UNIX_ENV is listed in the *Ada User's Guide*.

F 8.1.9 Potential Problems With I/O From Ada Tasks

In an Ada tasking environment on the HP 9000 Series 300, the Ada run time provides some protection of file objects against attempts to perform multiple simultaneous I/O operations on the same logical (internal) file. (Note that this is *not* the same as the sharing of an external file through multiple internal files which is supported in a controlled fashion by the use of the FORM parameter in CREATE or OPEN calls.) When multiple tasks attempt to share an internal file without proper synchronization, intermixing of two I/O operations on the same internal file (an Ada file object) can occur due to task scheduling. If such a situation exists and two I/O operations collide, PROGRAM_ERROR will be raised by the Ada I/O package.

CAUTION

It is your responsibility to utilize proper synchronization and mutual exclusion in the use of shared resources. Note that shared access to a common resource (in this case, a file) could be achieved by rendezvous between tasks that share that resource. If you write a program in which two tasks attempt to perform I/O operations on the same logical file without proper synchronization, that program is erroneous. (See *LRM*, Section 9.11.)

F 8.1.10 I/O Involving Symbolic Links

Some caution must be exercised when using an Ada program that performs I/O operations to files that involve symbolic links. (For more detail on the use of symbolic links to files in HP-UX, see *ln(1)*.)

Creating a symbolic link to a file creates a new name for that file that is, in effect, an alias for the actual file name. If you use the actual file name or its alias (that is, the name involving symbolic links), Ada I/O operations will work correctly. However, the `NAME` function (in the `TEXT_IO`, `SEQUENTIAL_IO`, and `DIRECT_IO` packages) will always return the actual rooted path of a file and *not* a path involving symbolic links.

F 8.2 The FORM Parameter

For both the CREATE and OPEN procedures in *Ada*, the FORM parameter specifies the characteristics of the external file involved.

The CREATE procedure establishes a new external file of a given NAME and FORM, and associates with it a specific program FILE object. The external file is created (and the FILE object set) with a certain file MODE. If the file already exists with the same NAME as stated in the CREATE call, the file will be erased and then recreated. The exception USE_ERROR is raised if the file mode is IN_FILE.

If you execute an Ada program containing a CREATE statement, you must have adequate permission for file creation in HP-UX. Otherwise, the CREATE procedure will raise STATUS_ERROR or USE_ERROR on attempts to recreate (erase) an existing file (regardless of the FORM parameter setting permissions). For example, an existing file with no read, write, or execute permission for the user who executes such an Ada program (one which attempts to create a file by the same name), will raise STATUS_ERROR or USE_ERROR.

The OPEN procedure associates an existing external file of a given NAME and FORM, with a specified program FILE object. The procedure also sets the current FILE mode. If there is an inadmissible change of MODE, an Ada USE_ERROR is generated.

F 8.2.1 An Overview of FORM Attributes

The FORM parameter is composed from a list of attributes that specify

- File protection
- File sharing
- File structuring
- Buffering
- Appending
- Blocking
- Terminal Input

F 8.2.2 The Format of FORM Parameters

Attributes of the FORM parameter have a keyword followed by the Ada "arrow symbol" (=>) (see *LRM*, Section 2.2(10)), followed by qualifier. The qualifier and arrow symbol may sometimes be omitted. Thus, the format for an attribute specifier is

```
KEYWORD
```

or

```
KEYWORD => QUALIFIER
```

The general format for the FORM parameter is a string formed from a list of attributes, with attributes separated by commas (,). The string is not case sensitive. The arrow symbol can be separated by spaces from the keyword and qualifier. The two forms below are equivalent:

KEYWORD => QUALIFIER

KEYWORD=>QUALIFIER

In some cases, an attribute can have multiple qualifiers that can be presented at the same time. In cases that allow multiple qualifiers, additional qualifiers are introduced with an underscore (_). Note that spaces are not allowed between the additional qualifiers; only underscore characters are allowed. Otherwise, a USE_ERROR exception is raised by CREATE. The two examples that follow illustrate the FORM parameter format.

The first example illustrates the use of the FORM parameter in the TEXT_IO.OPEN to set the file buffer size.

```
-- Example of opening a file  using the non-generic package TEXT_IO
-- This illustrates the use of the FORM parameter BUFFER_SIZE
-- Note: "inpt_file" must exist, or NAME_ERROR will be raised.
--
with TEXT_IO;
procedure STEST is

  TFILE : TEXT_IO.FILE_TYPE; --Define a file object for reference in Ada

begin -- STEST

   TEXT_IO.OPEN (FILE => TFILE,      -- Refer to file object created above
                 MODE => TEXT_IO.IN_FILE, --Set mode for READ access only
                 NAME => "inpt_file", -- Associate an external file name
                                   -- "inpt_file" to the file object: TFILE
                 FORM => "BUFFER_SIZE =>4096"--set buffer size to 4K bytes
                 );

end STEST;
```

The second example illustrates the use of the FORM parameter in TEXT_IO.CREATE. This example sets the access rights of the owner (HP-UX file permissions) on the created file and shows multiple qualifiers being presented at the same time.

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"owner=>read_write_execute");
```

F 8.2.3 The FORM Parameter Attribute - File Protection

The file protection attribute is only meaningful for a call to the CREATE procedure.

File protection involves two independent classifications. The first classification specifies *which user* can access the file and is indicated by the keywords listed in Table F-5.

Table F-5. User Access Categories

Category	Grants Access To
OWNER	Only the owner of the created file.
GROUP	Only the members of a defined group.
WORLD	Any other users.

Note that WORLD is similar to "others" in HP-UX terminology, but was used in its place because OTHERS is an Ada reserved word.

The second classification specifies *access rights* for each classification of user. The four general types of access rights, which are specified in the FORM parameter qualifier string, are listed in Table F-6.

Table F-6. File Access Rights

Category	Allows the User To
READ	Read from the external file.
WRITE	Write to the external file.
EXECUTE	Execute a program stored in the external file.
NONE	The user has no access rights to the external file. (This qualifier overrides any prior privileges.)

More than one access right can be specified for a particular file. Additional access rights can be indicated by separating them with an underscore as noted earlier. The following example using the FORM parameter in TEXT_IO.CREATE sets access rights of the owner and other users (HP-UX file permissions) on the created file. This example illustrates multiple qualifiers being used to set several permissions at the same time.

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"owner=>read_write_execute, world=>none");
```

Note that the HP-UX umask(1) and umask(2) commands may have set the default rights for any unspecified permissions. In the previous example, permission for the users in the category GROUP were unspecified. Typically, the default umask will be set so that the default allows newly created files to have read and write permission - and no execute permission - for each category of user (USER, GROUP, and WORLD).

Consider the case where the users in WORLD want to execute a program in an external file, but only the owner may modify the file.

The appropriate FORM parameter is then:

```
WORLD => EXECUTE,

OWNER => READ_WRITE_EXECUTE
```

This would be applied as:

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"world=>execute, owner=>read_write_execute");
```

Repetition of the same qualifier within attributes is illegal:

```
WORLD => EXECUTE_EXECUTE                          -- NOT legal
```

But repetition of entire attributes is allowed:

```
WORLD => EXECUTE, WORLD => EXECUTE          -- legal
```

F 8.2.4 The FORM Parameter Attribute - File Sharing

The file sharing attribute of the FORM parameter provides control over multiple access to a given external file from within an Ada program. For example, the controlled file sharing could be among separate Ada tasks. The control over multiple access does not extend to the whole system. See Section F 8.1.5 for details on file sharing at the system level between separate programs for more details. The HP-UX operating system controls file sharing at the system level; the programmer controls the file sharing described here. File sharing with the FORM parameter occurs within a single Ada program.

An external file can be shared; that is, the external file can be associated simultaneously with several logical file objects created by the OPEN or CREATE procedures. The file sharing attribute restricts or suppresses this capability by specifying one of the modes listed in Table F-7.

Table F-7. File Sharing Attribute Modes

Mode	Description
NOT_SHARED	Indicates exclusive access. No other logical file can be associated with the external file.
SHARED=>READERS	Only logical files of mode IN can be associated with the external file.
SHARED=>SINGLE_WRITER	Only logical files of mode IN and at most one file with mode OUT can be associated with the external file.
SHARED=>ANY	No restrictions; this is the default.

A USE_ERROR exception is raised if either of the following conditions exists for an external file already associated with a logical Ada file:

- A further OPEN or CREATE specifies a file sharing attribute different from the current one.
- A further OPEN, CREATE, or RESET violates the conditions imposed by the current file sharing attribute.

The restrictions imposed by the file sharing attribute disappear when the last logical file linked to the external file is closed. The default ANY allows multiple writers to a file. Specifying ANY can produce an unexpected sequence of output operations. Tasking programs should use synchronization via rendezvous to prevent problems that may arise in accessing a shared file. (See Section F 8.1.9.)

F 8.2.5 The FORM Parameter Attribute - File Structuring

This section describes how to structure your files.

F 8.2.5.1 Text Files

There is no FORM parameter to define the structure of text files. A text file consists of a sequence of bytes containing ASCII character codes.

You can control Ada terminators explicitly in text files. The representation of Ada terminators depends on the file's mode (IN or OUT) and whether it is associated with a terminal device or a mass-storage file. See Table F-8 which follows.

Table F-8. Text File Terminators

File Type	Terminator
Mass-storage files	end of line: ASCII.LF end of page: ASCII.LF ASCII.FF end of file: ASCII.LF ASCII.EOT
Terminal device/ IN mode	end of line: ASCII.LF end of page: ASCII.LF ASCII.FF end of file: ASCII.LF ASCII.FF
Terminal device/ OUT mode	end of line: ASCII.LF end of page: ASCII.FF end of file: ASCII.EOT

See Section F 8.1.3.3 for more information about terminators in text files.

F 8.2.5.2 Binary Files

This section describes use of the FORM parameter for binary (sequential or direct access) files. Two FORM attributes, RECORD_SIZE and RECORD_UNIT, control the structure of binary files.

Such a file can be viewed as a sequence or a set of consecutive RECORDS. The structure of a record is

    [HEADER]   OBJECT   [UNUSED_PART]

A record is composed of up to three items:

1. A HEADER consisting of two fields (each of 32 bits)

- The length of the object in bytes.
- The length of the descriptor in bytes.

2. An OBJECT with the exact binary representation of the Ada object in the executable program, possibly including an object descriptor.

3. An UNUSED_PART of variable size to permit full control of the record's size.

The HEADER is implemented only if the actual parameter of the instantiation of the I/O package is unconstrained.

The file structure attributes take the form:

```
RECORD_SIZE => size_in_bytes

RECORD_UNIT => size_in_bytes
```

The attributes' meaning depends on the object's type (constrained or unconstrained) and the file access mode (sequential or direct access).

The two types of access are:

- Sequential access of consecutive RECORDS
- Direct access of consecutive RECORDS

The consequences of this are shown in Table F-9 on the following page.

Table F-9. Structuring Binary Files With the FORM Parameter

Object Type	File Access Mode	RECORD_UNIT Attribute	RECORD_SIZE Attribute
Constrained	Sequential I/O	The RECORD_UNIT attribute is illegal.	If the RECORD_SIZE attribute is omitted, no UNUSED_PART is implemented. The default RECORD_SIZE is the object's size. If present, the RECORD_SIZE attribute must specify a record size greater than or equal to the object's size. Otherwise, the exception USE_ERROR is raised.
Unconstrained	Sequential I/O	By default, the RECORD_UNIT attribute is one byte. The size of the record is the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its length. This is the only case where record of a file can have different sizes.	The RECORD_SIZE attribute is illegal.
	Direct I/O	The RECORD_UNIT attribute is illegal.	The RECORD_SIZE attribute has no default value, and if a value is not specified, a USE_ERROR is raised. If you attempt to input or output an object larger than the given RECORD_SIZE, a DATA_ERROR exception is raised.

If you CREATE a direct access file with the default FORM parameter, then only objects of a constrained type can be written to or read from that direct access file.

F 8.2.6 The FORM Parameter Attribute - File Buffering

The buffer size can be specified by the attribute:

```
BUFFER_SIZE => size_in_bytes
```

The default value for BUFFER_SIZE is 0 for terminal devices (which means no buffering), and 1__sector for disc files. Using the file buffering attribute can improve I/O performance in some cases.

An example of the use of the FORM parameter in the TEXT_IO.OPEN to set the file buffer size is shown below:

```
-- Example of creating a file using the non-generic package TEXT_IO
-- This illustrates the use of the FORM parameter BUFFER_SIZE

with TEXT_IO;
procedure T_TEST is

  BFILE : TEXT_IO.FILE_TYPE; -- Define a file object for reference in Ada

begin -- T_TEST

  TEXT_IO.CREATE (FILE => BFILE, -- Refer to file object created above
                  MODE => TEXT_IO.OUT_FILE, -- Set WRITE access only mode
                  NAME => "txt_file",  -- Associate an external file name
                                       -- "txt_file" to the file object: BFILE
                  FORM => "BUFFER_SIZE=>8192"--set buffer size to 8K bytes
                 );

end T_TEST;
```

F 8.2.7 The FORM Parameter - Appending to a File

The APPEND attribute can only be used with the procedure OPEN. Its format is:

```
APPEND
```

Any output will be placed at the end of the named external file.

In normal circumstances, when an external file is opened, an index is set that points to the beginning of the file. If the APPEND attribute is present for a sequential or text file, data transfer commences at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

The APPEND attribute is *not* applicable to terminal devices.

F 8.2.8 The FORM Parameter Attribute - Blocking

This attribute has two alternative forms:

BLOCKING

or

NON_BLOCKING

This attribute specifies the I/O system behavior when a request for data transfer cannot be fulfilled at that moment. This stoppage may be due to the unavailability of data, or to the unavailability of the external file device.

F 8.2.8.1 Blocking

If the blocking attribute is set, the task waits until the data transfer is complete, and all other tasks are suspended (or blocked). The system is busy waiting.

The default for this attribute depends on the actual program. It is BLOCKING for programs without any task declarations and it is NON_BLOCKING for a program containing tasks.

F 8.2.8.2 Non-Blocking

If the non-blocking attribute is set, the task that ordered the data transfer is suspended and the other tasks can execute. The suspended task is rescheduled and kept in a ready state together with other tasks in a ready state at the same priority level.

When the suspended task is scheduled again, the data transfer request is reactivated. If ready, the transfer is activated; otherwise the rescheduling is repeated. Control returns to your program after the data transfer is complete.

F 8.2.9 The FORM Parameter - Terminal Input

The terminal input attribute takes one of two alternative forms:

```
TERMINAL_INPUT => LINES,

TERMINAL_INPUT => CHARACTERS,
```

Terminal input is normally processed in units of one line at a time, where a line is delimited by a special character. A process attempting to read from the terminal as an external file is suspended until a complete line has been typed. At that time, the outstanding read call (and possibly also later calls) is satisfied.

The LINES option specifies line-at-a-time data transfer, which is the default case.

The CHARACTERS option means that data transfers character by character, and so a complete line does not have to be entered before the read request can be satisfied. For this option, the BUFFER_SIZE must be zero.

The TERMINAL_INPUT attribute is only applicable to terminal devices other than the one pre-connected to STANDARD_INPUT and STANDARD_OUTPUT.

F.9 The HP Ada Compilation System and HP-UX Signals

The HP Ada run time in the HP-9000 Series 300 uses HP-UX signals to implement the following featur of the Ada language:

- Ada exception handling
- Ada task management
- Ada delay timing

F 9.1 HP-UX Signals Reserved by the Ada Run Time

The HP-UX signals reserved by the Ada run time are:

- SIGFPE
- SIGILL
- SIGBUS
- SIGSEGV
- SIGEMT
- SIGALRM
- SIGVTALRM

WARNING

The signals reserved for use by the HP Ada run time may be used in your Ada programs only with extreme caution. If you attempt to generate, catch, or ignore these signals, the Ada program can produce unpredictable results.

F 9.2 HP-UX Signals Used for Hp Ada Exception Handling

Signals are used to raise some HP Ada exceptions. The HP Ada run time's handlers for operating syster signals are set during the elaboration of the HP Ada run time. Defining a new handler for any of thes signals destroys the normal exception handling mechanism of Ada and may result in erroneous run-tim execution.

The following signals are used for exception handling:

```
SIGILL, SIGFPE, SIGBUS, SIGSEGV, SIGEMT
```

F 9.3 HP-UX Signals Used for Ada Task Management

The HP-UX alarm facility is used by the Ada run time for task management. When the Ada prograr contains tasks, the HP-UX alarm signal SIGVTALRM is used to implement time slicing. In time slicing, th Ada run time allocates the available processor time among concurrent tasks. If the Ada program does no contain tasks, it is a sequential program. The Ada run time in sequential programs does not use th HP-UX signal SIGVTALARM.

indicate the reason for termination (if caused by a signal) in the exit status. The HP-UX convention is adhered to only for the signals listed in Section F 9.6 (which cause termination). For other signals, the exit status is non-zero but does not indicate that the cause of termination was a signal.

F 9.8 Programming in Ada With HP-UX Signals

If you intend to utilize signals in interfaced subprograms, refer to the section on "Potential Problems Using Interfaced Subprograms" (Section F 14.7). This version of the product does not support the association of an HP-UX signal such as SIGINT with an Ada procedure or a task entry. The signal must be handled inside an external subprogram. The same cautions apply for this external subprogram as for any external interfaced subprogram that might be interrupted by an unexpected signal.

Table F-10 provides a summary of HP-UX Signals.

F 10. Predefined Language Attributes

The predefined language attributes are implemented as defined in the *Reference Manual for the Ada Programming Language*, Appendix A, with the following exception.

```
P'STORAGE_SIZE
```

When applied to an access type or subtype of an access type the compiler produces one of two responses:

- If there is no length clause, the compiler issues the warning:

```
The prefix of the attribute STORAGE_SIZE is an access type which has no length
clause. The value returned by STORAGE_SIZE is always 0.
```

- If a length clause is specified, the compiler issues the error:

```
Length clauses are not currently implemented.
```

F 11. Predefined Language Pragmas

The predefined language pragmas are implemented as defined in the *Reference Manual for the Ada Programming Language*, Appendix B, with the following exceptions (see also "Implementation-Dependent Pragmas" in this appendix).

Use of the predefined language pragmas

```
CONTROLLED
MEMORY_SIZE
OPTIMIZE
PACK
SHARED
STORAGE_UNIT
SYSTEM_NAME
```

is either ignored without comment (the pragma has no effect as required by the *Reference Manual for the Ada Programming Language*) or elicits this warning from the compiler:

```
This pragma is not currently supported by the implementation.
```

F 12. Data Type Ranges, Alignment, and Layout

Tables F-11 and F-12 show data type ranges and alignment. Information on layout of data objects in memory is contained in Section 12.3.

F 12.1 Data Type Ranges

Table F-11 shows data type ranges.

Table F-11. Ranges

Ada Type	Range
SHORT_SHORT_INTEGER	-128 .. 127
SHORT_INTEGER	-32768 .. 32767
INTEGER	-2_147_483_648 .. 2_147_483_647
ENUMERATION	32768 elements maximum. 0..127 (for <= 128 elements) or 0..32767 (for > 128 elements)
BOOLEAN	FALSE..TRUE (2#0000_0000#, 2#1111_1111#)
CHARACTER	ASCII.NUL..ASCII.DEL' (0..127)
FLOAT	-2#1.111_1111_1111_1111_1111_1111#E+127 .. 2#1.111_1111_1111_1111_1111_1111#E+127 (that is,-3.402823E+38..+3.402823E+38, approximately)
LONG_FLOAT	-2#1.1111_1111_1111_1111_1111_1111_1111-1111_1111_1111_1111_1111_1111#E+1023 ..2#1.1111_1111_1111_1111_1111_1111_1111-1111_1111_1111_1111_1111_1111#E+1023 (that is, -1.797693134862315E+308 .. +1.797693134862315E+308, approximately)

F 12.2 Data Alignment

Table F-12 lists data alignment.

Table F-12. Alignment

Ada Type	Alignment
SHORT_SHORT_INTEGER	allocated on a byte boundary
SHORT_INTEGER	allocated on a half word boundary
INTEGER	allocated on a word boundary
ENUMERATION (<= 128 elements)	allocated on a byte boundary
ENUMERATION (> 128 elements)	allocated on a half-word boundary
BOOLEAN	allocated on a byte boundary
CHARACTER	allocated on a byte boundary
FIXED POINT	mapped into one of the three integer types
FLOATING POINT	allocated on a word boundary
COMPOSITE	may be allocated on a byte or word boundary depending on the alignment constraints of the components or elements
ACCESS	allocated on a word boundary
TASK	allocated on a word boundary

F 12.3 Layout of Data Objects in Memory

1. Integer Types

SHORT_SHORT_INTEGER, SHORT_INTEGER, and INTEGER are respectively allocated 8 bits, 16 bits, and 32 bits. For a user-defined integer type, the representation chosen is the smallest of the predefined integer types whose range includes the range of the declared type.

2. Enumeration Types Other Than Boolean

A value of an enumeration type without a representation clause (*LRM*, Section 13.3) is represented by the non-negative integer value corresponding to its position in the type definition (positions are numbered from zero). The values of enumeration types are signed (but never negative). Thus, only enumeration types defining up to 128 elements are represented in a byte; others are represented in 16 bits (a half word).

3. The Type CHARACTER

The type CHARACTER is represented as a user-defined enumeration type of less than 129 elements (integer range 0 through 127), corresponding to ASCII.NUL through ASCII.DEL.

4. The Type BOOLEAN

The values FALSE and TRUE are represented as the 8-bit value 0 and 255 (that is, 2#1111_1111#) zero and one in a byte. The value of BOOLEAN'POS (TRUE) is one and the value of BOOLEAN'POS (FALSE) is zero. The result type is universal_integer for the POS attribute.

5. Fixed Point Types

A value of a fixed point type is managed by the compiler as the value of signed_mantissa * small. Signed_mantissa is a signed integer. Small is the largest power of two that is not greater than the delta of the fixed accuracy definition. Small is a compile-time constant, so signed_mantissa is the only portion that needs memory space at run time. Thus, fixed point types are mapped into one of the three predefined integer types.

6. Floating Point Types

A floating point value of type FLOAT is represented in normalized 32-bit IEEE format:

bit 0: sign (most significant bit)

bits 1 to 8: exponent

bits 9 to 31: mantissa

A floating point value of type LONG_FLOAT is represented in normalized 64-bit IEEE format:

bit 0: sign (most significant bit)

bits 1 to 11: exponent

bits 12 to 63: mantissa

7. Array Types

- General case: array elements in Ada are allocated contiguously in memory and stored by rows. This is called row-major order, with the last subscript varying most rapidly, the next to last varying next most rapidly, and the first subscript varying least rapidly. The size of an array is the product of the number of its elements and its element size.

- Special case: elements with an odd number of bytes are padded to start at a word boundary. An example of this is an array with an element type that is a record. The record contains two components, of type INTEGER and BOOLEAN, respectively. Instead of allocating these record type elements contiguously, all elements are extended with a null byte to make them even sized. Therefore, the array size (in bytes) is the product of the number of elements and the number represented by one greater than the element size (in bytes).

8. Record Types

Record components are allocated contiguously in memory with holes inserted between components to satisfy their alignment constraints. The alignment constraint of a record is the strongest alignment constraint of any of its components.

Implicit components are components created by the compiler within a record. They can be used to hold the record size, array or record descriptors, or variant indexes. The implicit component for the record size is, in general, placed at the beginning of the record.

Ada semantics do not require a particular ordering of record components. Record components can be rearranged by the compiler to take into account alignment constraints. Record components can be arranged in an order other than the declarative order.

For dynamic components, a *dope* (pointer) is created where the component would normally have been placed, and the component itself is allocated at the end of the record. The dope contains the offset from the start of the record to the actual component.

If the size of a component depends upon a discriminant, the maximum size is allocated.

For a record type with variant parts, the common part and each variant part are laid out in the same way as described above, with the exception that dynamic components are laid out at the end of the entire record, not at the end of a specific variant part. Variant parts are overlaid when possible and holes of previously laid out variant parts are used whenever possible.

9. Parameters

When passing parameters by value within HP Ada, byte-aligned objects are considered as word aligned and right-justified and odd-byte-sized parameters are passed in word-sized containers. This ensures that the stack pointer remains divisible by 4. (For information on passing parameters to external interfaced subprograms written in other languages, refer to Section F 14.1 of this appendix.)

F 13. Compiler Limitations

NOTE

It is impossible to give exact numbers for most of the limits given here. The various language features may interact in complex ways to lower the following limits. The numbers represent "hard" limits in simple program fragments devoid of other Ada features.

Limit	Description
255	Maximum number of characters in a source line.
253	Maximum number of characters in a string literal.
255	Maximum number of characters in an enumeration type element.
32767	In an enumeration type, the sum of the lengths of the IMAGE attributes of all elements in the type, plus the number of elements in the type, must not exceed this value.
2047	Maximum number of actual compilation units in a library.
32767	Maximum number of enumeration elements in a single enumeration type (this limit is further constrained by the maximum number of characters for all enumeration literals of the type).
2047	Maximum number of "created" units in a single compilation.
2**31-1	Maximum number of bits in any size computation.
2048	Links in a library.
2048	Libraries in the INSTALLATION family (250 of which are reserved).
2047	Libraries in either the PUBLIC or a user defined family. (For more information, see the *Ada User's Guide*, which discusses families of Ada libraries and the supported utilities (tools) to manage them).
-	Maximum number of tasks is limited only by heap size.
-	The offset of a record component has to be representable in 16-bit integer.

The following items are limited only by overflow of internal tables (AIL or HLST tables). All internal data structures of the compiler which previously placed fixed limits are now dynamically created.

- Maximum number of identifiers in a unit. An identifier includes enumerated type identifiers, record field definitions, and (generic) unit parameter definitions.
- Maximum "structure" depth. Structure includes the following: nested blocks, compound statements, aggregate associations, parameter associations, subexpressions.
- Maximum array dimensions. Set to Maximum structure depth/10.*
- Maximum number of discriminants in a record constraint.*
- Maximum number of associations in a record aggregate.*
- Maximum number of parameters in a subprogram definition.*
- Maximum expression depth.*
- Maximum number of nested frames. Library-level unit counts as a frame.
- Maximum number of overloads per compilation unit.
- Maximum number of overloads per identifier.

* A limit on the size of tables used in overloading resolution can potentially lower this figure. This limit is set at 500. It reflects the number of possible interpretations of names in any single construct under analysis by the compiler (procedure call, assignment statement, etc.).

The following limits apply to the Ada development environment (`ada.umgr`, `ada.fmgr`, Ada tools).

Limit	Description
200	The number of characters in the actual rooted path of an Ada program `LIBRARY` or `FAMILY` of libraries.
200	The number of characters in the string (possibly after expansion by an HP-UX shell) specifying the name of an Ada program `LIBRARY` or `FAMILY` of libraries. This limit applies to strings (pathname expressions) specified for a `LIBRARY` or `FAMILY` that you submit to tools such as `ada.mklib` or `ada.umgr`.
512	Maximum length of an input line for the tools `ada.fmgr` and `ada.umgr`.

F 14. Calling External Subprograms From Ada

In the HP implementation of Ada, subprogram parameters of external interfaced subprograms are passed on the stack in *reverse* order of their declaration. The first parameter appears at the top of the stack. This same ordering is used by HP for other language products on the HP 9000 Series 300 family of computers. The languages described in Section 14 of this Appendix are HP implementations of 680X0 Assembler, C, FORTRAN 77, and Pascal on the HP-UX Series 300 systems.

In some cases, the interface requires not only the parameters but additional information to be pushed on the stack as well. This information may include a parameter count, a return result pointer, or other bookkeeping information. When you specify the interfaced language name, that name is used to select the correct calling conventions for supported languages. Then, subprograms written in HP C, HP FORTRAN 77, and HP Pascal interface correctly with the HP Ada subprogram caller.

Section F 14 contains detailed information about calling subprograms written in these languages. If the subprogram is written in a language from another vendor, you must follow the standard calling conventions.

In the HP Ada implementation of external subprogram interfaces, the three Ada parameter passing modes (IN, OUT, IN OUT) are supported, with some limitations as noted below. Scalar and access parameters of mode IN are passed by *value*. All other parameters of mode IN are passed by *reference*. Parameters of mode OUT or IN OUT are always passed by reference. (See Table F-13 and Figure F-1.)

Table F-13. Ada Types and Parameter Passing Modes

Ada Type	Mode Passed By Value	Mode Passed By Reference
SCALAR, ACCESS	IN	OUT, IN OUT
All others except TASK, PRIVATE, and FIXED POINT (REAL)		IN, OUT, IN OUT
TASK, PRIVATE, and FIXED POINT (REAL)	(not passed)	(not passed)

The values of the following types can*not* be passed as parameters to an external subprogram:
Task types (*LRM* Section 9.1, 9.2),
Private types (*LRM* Section 7.4),
Fixed point types (*LRM* Section 3.5.9, 3.5.10).

A composite type (an array or record type) is always passed by reference (as noted above). A component of a composite type is passed according to its type classification (scalar, access, or composite).

Only scalar types (enumeration, character, boolean, integer, or real) or access types are allowed for the result returned by an external function subprogram.

NOTE

There are no checks for consistency between the subprogram parameters (as declared in Ada) and the corresponding external subprogram parameters. Because external subprograms have no notion of Ada's parameter modes, parameters passed by reference are not protected from modification by an external subprogram. Even if the parameter is declared to be only of mode IN (and not OUT or IN OUT) but is passed by reference (that is, an array or record type), the value of the Ada actual parameter can still be modified.

The possibility that the parameter's actual value will be modified by an interfaced external subprogram exists when that parameter is not passed by value. Objects whose attribute 'ADDRESS is passed as a parameter and parameters passed by reference are not protected from alteration and are subject to modification by the external subprogram. In addition, such objects will have no run-time checks performed on their values upon return from interfaced external subprograms.

Erroneous results may occur if the parameter values are altered in some way that violates Ada constraints for the actual Ada parameter. The responsibility to ensure that values are not modified in external interfaced subprograms in such a manner as to subvert the strong type and range checking enforced by the Ada language is yours.

CAUTION

Be very careful to establish the exact nature of the types of parameters to be passed. The bit representations of these types can be different between this implementation of Ada and other languages, or between different implementations of the same language. For example, between different releases of HP Ada, the redefinition of the size of the predefined standard type INTEGER may have changed, increasing from a 16-bit storage size to a 32-bit storage size. Stacked values must occupy equal space in the two interfaced languages. When passing record types, pay particular attention to the internal organization of the elements of a record because Ada semantics do not guarantee a particular order of components. Moreover, Ada compilers are free to rearrange or add components within a record. (See Section F 12 on data type layout for more information.)

F 14.1 General Considerations in Passing Ada Types

Section F 14.1 discusses each data type in general terms. Sections F 14.2 through F 14.5 describe the details of interfacing your Ada programs with external subprograms written in HP C, HP FORTRAN 77, HP Pascal. Section F 14.6 provides summary tables.

The Ada types are described in the following order:

- Scalar
    - Integer
    - Enumeration
    - Boolean
    - Character
    - Real
- Access
- Array
- Record
- Task
- Private

F 14.1.1 Scalar Types

This section describes general considerations when you are passing scalar types between Ada programs and subprograms written in a different HP language. The class *scalar types* includes integer, real, and enumeration types. Because character and boolean types are predefined Ada enumeration types, they are also scalar types.

Scalar type parameters of mode IN are passed by value. Scalar type parameters of mode IN OUT or OUT are passed by reference.

F 14.1.1.1 Integer Types

In HP Ada, all integers are represented in two's complement form. SHORT_SHORT_INTEGERS are represented as 8-bit numbers, SHORT_INTEGERS are represented as 16-bit numbers, and INTEGERS are represented as 32-bit numbers.

All integer types can be passed to interfaced subprograms. When an integer is used as a parameter for an interfaced subprogram, the call may be made either by reference or by value. For a call by reference, the value of the actual integer parameter is not copied or modified, but a 32-bit address pointer is pushed on the stack. For a call by value, a copy of the actual integer parameter value is pushed on the stack, with sign extension as necessary to satisfy the requirements of the external subprogram. See Sections F 14.2.1.1, F 14.3.1.1, F 14.4.1.1, and F 14.5.1.1 for details specific to interfaced subprograms written in different languages.

Integer types may be returned as function results from external interfaced subprograms.

F 14.1.1.2 Enumeration Types

Values of an enumeration type (*LRM* 3.5.1) without an enumeration representation clause (*LRM* 13.3) have an internal representation of the value's position in the list of enumeration literals defining the type. These values are non-negative. The first literal in the list corresponds to an integer value of zero.

There can be no more than 32768 elements in an enumeration type and, because the program representation of the position number of an element starts at zero, the maximum value is 32767.

Values of enumeration types with less than 128 elements are represented as a non-negative integer in eight bits, whereas values of other enumeration types are represented as a non-negative integer in 16 bits. When passed by value, the copy of the integer is treated as if it were an unsigned integer, with any necessary extension up to 32 bits to satisfy the requirements of the external subprogram. See Sections F 14.2.1.2, F 14.3.1.2, F 14.4.1.2, and F 14.5.1.2 for specific details.

Enumeration types may be returned as function results from external interfaced subprograms.

F 14.1.1.3 Boolean Types

The values of the predefined enumeration type BOOLEAN are represented in HP Ada as 8-bit values. The boolean value FALSE is represented by the 8-bit value zero, and the boolean value TRUE is represented by the 8-bit value 255 (#1111_1111#). This is *not* the same as other enumeration types that occupy eight bits in Ada, where the position in the enumeration list at declaration defines the internal representation for the element.

When a boolean is passed by reference, its value is not copied, but a 32-bit address pointer is pushed on the stack. When a boolean is passed by value, a copy is pushed on the stack. The Ada boolean value occupies one byte, and it is up to you to ensure that the boolean is in a correct format for the external subprogram.

Boolean types may be returned as function results from external interfaced subprograms, with caution as noted above.

See Sections F 14.2.1.3, F 14.3.1.3, F 14.4.1.3, and F 14.5.1.3 for information about interfaced subprograms written in Assembler, HP C, HP FORTRAN 77, and HP Pascal.

F 14.1.1.4 Character Types

The values of the predefined enumeration type CHARACTER are represented as 8-bit values in a range 0 through 127.

character types may be returned as function results from external subprograms.

F 14.1.1.5 Real Types

Ada fixed point types and Ada floating point types are discussed in the following subsections.

Fixed Point Real Types

Ada fixed point types (*LRM* 3.5.9, 3.5.10) are *not* supported as parameters or as results of external subprograms.

Fixed point types cannot be returned as function results from external subprograms.

Floating Point Real Types

Floating point values (*LRM* Sections 3.5.7 and 3.5.8) in the HP implementation of Ada are of 32 bits (FLOAT) or 64 bits (LONG_FLOAT). These two types conform to the conventions defined in the document "A Proposed Standard for Binary Floating Point Arithmetic", IEEE P754, draft 10.0.

When passed by reference, a 32-bit address pointer to the object is pushed on the stack. When passed by value, a copy is pushed on the stack, possibly with extension to 64 bits according to the external subprogram interfacing conventions. See Sections F 14.2.1.5, F 14.3.1.5, F 14.4.1.5, and F 14.5.1.5 for details specific to interfaced subprograms written in different languages.

Floating point types may be returned as function results from external interfaced subprograms, with some restrictions.

F 14.1.2 Access Types

Values of an access type (*LRM*, Section 3.8) have an internal representation as the 32-bit address of the underlying designated object (such as an object__address stored in an object__address__location). An access type's value is an address pointer to the object. Therefore, when an access type is passed by value, a copy of this 32-bit object__address is pushed on the stack. If the type is passed by reference, however, the 32-bit address pointer to the object__address__location is pushed on the stack. This is effectively a double indirect address to the designated object. See Figure F-1.

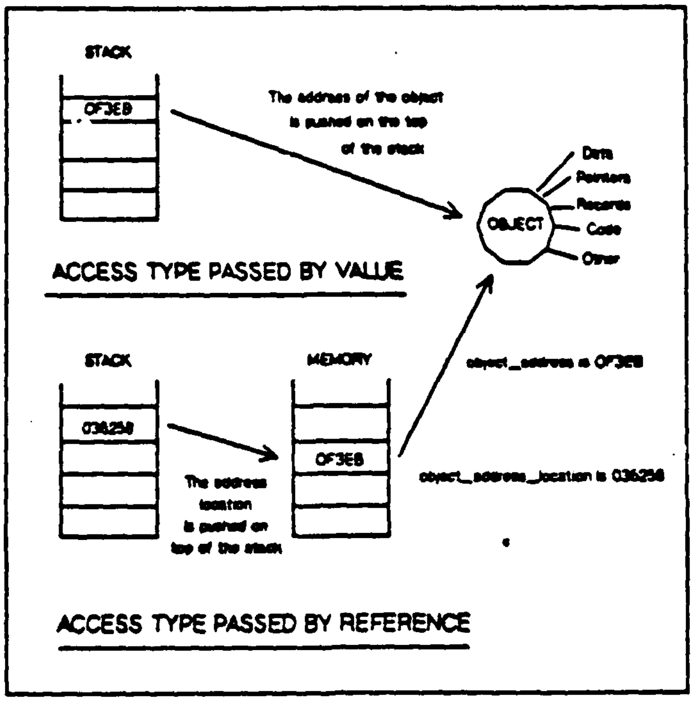

Figure F-1. Passing Access Types to Interfaced Subprograms

Access types may be returned as function results from external interfaced subprograms.

Ada access types are pointers to Ada objects. Ada access types to simple objects (scalar type or other access type objects) are actually pointers to the designated object. Ada access type values pointing to composite types (record or array types) are pointers to Ada descriptors for that object, rather than pointers to the actual object.

CAUTION

The descriptors used by Ada are *not identical* to descriptors used by other languages. Therefore, an Ada access type value that points to a composite type may have no meaning or may have a different meaning when passed as a parameter to a subprogram written in another language.

F 14.1.3 Array Types

In the HP implementation of Ada, arrays (*LRM*, Section 3.6) are always passed by reference. The value pushed on the stack is the address of the first element of the array. When an array is passed as a parameter to an external interfaced subprogram, the usual checks on the consistency of array bounds between the calling program and the called subprogram are *not* enforced. You are responsible for ensuring that the external interfaced subprogram keeps within the proper array bounds.

Values of the predefined type STRING (*LRM*, Section 3.6.3) are a special case of arrays and are passed as described above. The address of the first character in the string is pushed on the stack. Returning strings from an external interfaced subprogram to Ada requires special handling because the descriptors for string types may differ between Ada and the called subprogram written in other languages.

Array element allocation, layout, and alignment are described in Section F 12 of this appendix. Array types cannot be returned as function results from external interfaced subprograms.

F 14.1.4 Record Types

Records (*LRM* 3.7) are always passed by reference in the HP implementation of Ada, pushing the 32-bit address of the first component of the record on the stack.

Unlike arrays, however, the individual components of a record may have been reordered internally by the Ada compiler. This means that the implementation of record components can be in an order different from the declarative order. Ada semantics do not require a specific ordering of record components.

You can control the layout of individual record components by using record representation clauses. However, in the absence of a complete record specification, the exact internal structure of a record in memory cannot be known directly at the time of coding (see Section F 12). The attribute 'POS can be used to locate the offset of a record component with respect to the starting address of the record. By passing such a record component offset to a called subprogram (in addition to passing the record), Ada record components may be directly accessed by an interfaced subprogram. Offsets are relative to the starting address of the record. The starting address of the record is derived from the following:

- the record passed as a parameter (records are always passed by reference).
- the attribute 'ADDRESS of the record passed as a parameter.
- an access to the record passed as a parameter.

Direct assignment to a discriminant of a record is not allowed in Ada (*LRM*, Section 3.7.1). A discriminant cannot be passed as an actual parameter of mode OUT or IN OUT. This restriction applies equally to HP Ada subprograms and to external interfaced subprograms written in other languages. If an interfaced program is given access to the whole record (rather than individual components), that code should not change the discriminant value. Record element allocation, layout, and alignment are described in Section F 12.

In HP Ada, records are packed and variant record parts are overlaid; the size of the record is the longest variant part. If a record contains discriminants or composite components having a dynamic size, the compiler may add implicit components to the record. These components can include a variant index, record or component size, and portions of array descriptors.

Dynamic components and components whose size depends upon record discriminant values are implemented indirectly within the record by using descriptors.

Record types cannot be returned as function results from external interfaced subprograms.

F 14.1.5 Task Types

Task types *cannot* be passed to an external procedure or function as parameters in HP Ada. Task types *cannot* be returned as function results from external interfaced subprograms.

The HP implementation of tasking in Ada uses asynchronous HP-UX signals. The asynchronous signals SIGALRM and SIGVTALRM are used by the Ada run time in the HP 9000 Series 300 implementation. If an external subprogram called within a tasking Ada main subprogram is interrupted either by SIGALRM or SIGVTALRM while executing an HP-UX primitive, problems may arise. (See Section 9, "The HP Ada Compilation System and HP-UX Signals", for more information.) You must therefore disable SIGALRM and SIGVTALRM before any HP-UX primitive is called from within the external interfaced subprogram written in a language different from Ada. SIGALRM and SIGVTALRM must be re-enabled after completion of the execution of the primitive, and the SIGALRM and SIGVTALRM interrupt cycle should then be restarted using the HP-UX routine alarm.

An alternative method of protecting interfaced code from signals is described in the *Ada User's Guide* section on execution-time topics. Two procedures, SUSPEND_ADA_TASKING and RESUME_ADA_TASKING, from the HP supplied package UNIX_ENV can be used to access signal protection primitives inside Ada programs. Calls to these routines could be used within an Ada program to surround a critical section of Ada code or a call to external interfaced subprogram code that contains a critical section.

See Sections F 9 and F 14.7 for more detail on the suggested handling of these signals.

F 14.1.6 Private Types

Private types cannot be returned as function results from external interfaced subprograms.

F 14.2 Calling Assembly Language Subprograms

When calling interfaced assembly language subprograms, specify the named external subprogram in a compiler directive:

```
pragma INTERFACE (ASSEMBLER, Ada_subprogram_name);
```

Note that the language_type specification is ASSEMBLER and not ASSEMBLY. This description refers to the HP assembly language for the MC680x0 microprocessor family (68K assembly language) upon which the Series 300 family is based.

When calling interfaced 68K assembly language subprograms, scalar and access parameters of mode IN are passed by value; the value of the parameter object is copied and pushed on the stack. All other types of IN parameters (array, records) and parameters of mode OUT and IN OUT are passed by reference; the address of the parameter object is pushed on the stack.

When calling 68K assembly language subprograms, the processor scratch registers are considered to be A0, A1, D0, or D1. In external interfaced 68K assembly subprograms, processor registers D2 through D7, and A2 through A7 must be saved on entry and restored before returning to the Ada caller. The Ada compiler expects those registers to be unchanged across a call to an external interfaced subprogram. Only registers used in the called 68K assembly language subprogram must be saved and restored, but you are responsible for ensuring that all register contents (except for designated scratch registers) are unchanged.

The results returned by external function subprograms are expected to be in the register D0 if the result is scalar, or in register A0 if the result is an access value. LONG_FLOAT values that are represented as 64-bits are returned in two registers: D0 holds the low-level word and D1 holds the high level word.

Only scalar types (integer, floating point, character, boolean, and enumeration types) and access types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language.

For more information on 68K assembly language interfacing, see the *HP-UX Assembler Reference Manual and ADA Tutorial*, the *MC68020 32-Bit Microprocessor User's Manual*, and the *MC68881 Floating-Point Coprocessor User's Manual*.

F 14.2.1 Scalar Types and Assembly Language Subprograms

Scalar types include integer, real, and enumeration types. The predefined enumeration types CHARACTER and BOOLEAN are also scalar types. This section discusses things you should consider when passing scalar types to and from interfaced 68K assembly language subprograms.

F 14.2.1.1 Integer Types and Assembly Language Subprograms

When passed by value to an assembler subprogram, values of type INTEGER and SHORT_INTEGER are copied and pushed on the stack without alteration. An Ada INTEGER is stacked in a 32-bit container (a Long Word in 68K assembly language) and an Ada SHORT_INTEGER is stacked in a 16-bit container (a Word in 68K 68K assembly language). Values of type SHORT_SHORT_INTEGER are copied and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit word; the low order 8 bits are meaningless and should not be accessed.

When passed by reference, the value is not altered and a 32-bit address pointer to the integer object is pushed on the stack.

Table F-14 summarizes the integer correspondence between Ada and HP assembly language.

Table F-14. HP Ada/68K Assembly Language Integer Correspondence

Ada	68K Assembly Language	Bit Length
SHORT_SHORT_INTEGER	Byte	8
SHORT_INTEGER	Word	16
INTEGER	Long Word	32

All integer types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language. The results returned by external function subprograms are expected to be in the register D0 if the result is an integer type, or in register A0 if the result is an access to an integer value.

F 14.2.1.2 Enumeration Types and Assembly Language Subprograms

When passed by value to an assembler subprogram, values of enumeration types represented by 16 bits are copied and pushed on the stack without alteration. Values of enumeration types represented by 8 bits are copied and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit word; the low order 8 bits are meaningless and should not be accessed.

When passed by reference, the value is not altered and a 32-bit address pointer to the enumeration object is pushed on the stack.

Enumeration types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language. The results returned by external function subprograms are expected to be in the register D0 if the result is an enumeration type, or in register A0 if the result is an access to an enumeration value.

F 14.2.1.3 Boolean Types and Assembly Language Subprograms

Although a predefined enumeration type, Ada boolean values are represented in 8 bits in a manner different from other enumeration types in the HP implementation of Ada . The internal representation of FALSE corresponds to 2#0000_0000# and TRUE corresponds to 2#1111_1111# (that is, all zeros or all ones in 8 bits). Note that the value of BOOLEAN'POS(TRUE) is 1 and the value of BOOLEAN'POS(FALSE) is 0.

When passed by value to an interfaced 68K assembly language subprogram, values of type BOOLEAN are copied, and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit word; the low order 8 bits are meaningless and should not be accessed.

When passed by reference to an interfaced 68K assembly language subprogram, the boolean value is not altered and a 32-bit address pointer to the object is pushed on the stack.

Boolean types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language, when care is taken to observe the internal representation. The results returned by external function subprograms are expected to be in the register D0 if the result is a boolean type, or in register A0 if the result is an access to a boolean value.

F 14.2.1.4 Character Types and Assembly Language Subprograms

Character types, which are predefined enumeration types, are represented as 8 bits in the HP implementation of Ada. When passed by value to an interfaced subprogram written in 68K assembly language, values of type CHARACTER are pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit word; the low order 8 bits are meaningless and should not be accessed.

When passed by reference to an interfaced subprogram written in 68K assembly language, the character values are not altered and a 32-bit pointer to the address of the character object is pushed on the stack.

Character types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language. The results returned by external function subprograms are expected to be in the register D0 if the result is a character type, or in register A0 if the result is an access to a character value.

F 14.2.1.5 Real Types and Assembly Language Subprograms

When passed by value to interfaced subprograms written in 68K assembly language, values of type FLOAT and LONG_FLOAT are copied and pushed on the stack without alteration. Specifically, Ada objects of type FLOAT are copied as a 68K assembler Long Word (32 bits) and Ada LONG_FLOAT objects are copied as two 68K assembler Long Words (64 bits).

When passed by reference to an interfaced subprogram written in 68K assembly language, the values are not altered and a 32-bit address of the object is pushed on the stack.

Floating point types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language. The results returned by external function subprograms are expected to be in the register D0 if the result is a FLOAT type. If the results returned by external function subprograms are LONG_FLOAT type, the results are expected to be contained in the register D0 and D1. The low-level word is in D0 and the high-level word is in D1. The result will be returned in register A0 if the result is an access to a FLOAT or LONG_FLOAT value.

F 14.2.2 Access Types and Assembly Language Subprograms

See general comments on parameter passing in Section F 14.1.2.

Access types are allowed for the result returned by an external interfaced subprogram written in 68K assembly language. The results returned by external function subprograms are expected to be in the register A0 if the result is an access value.

F 14.2.3 Array Types and Assembly Language Subprograms

See general comments on parameter passing in Section F 14.1.3.

Array types cannot be returned as function results from external interfaced subprograms.

F 14.2.4 Record Types and Assembly Language Subprograms

See general comments on parameter passing in Section F 14.1.4.

Record types cannot be returned as function results from external interfaced subprograms.

F 14.2.5 Task Types and Assembly Language Subprograms

Task types cannot be passed as parameters in HP Ada. Task types cannot be returned as function results from external subprograms.

F 14.2.6 Private Types and Assembly Language Subprograms

Private types cannot be passed as parameters in HP Ada. Private types cannot be returned as function results from external interfaced subprograms.

F 14.3 Calling HP C Subprograms

When calling interfaced HP C subprograms, the form

```
pragma INTERFACE (C, Ada_subprogram_name)
```

is used to identify the need for HP C parameter passing conventions.

To call the following HP C subroutine (assuming call by value parameter passing):

```
void c_sub (parm)
int parm;
{
   ...
}
```

Ada requires an interfaced subprogram declaration:

```
procedure C_SUB (PARM1 : in INTEGER);
pragma INTERFACE (C, C_SUB);
```

The external name specified in the Ada interface declaration can be any Ada identifier. No special handling of leading underscores is required as this is handled by the compiler to conform to standard calling conventions. Pragma INTERFACE ensures that the underscore required in front of the HP C subroutine name is correctly prepended by the compiler. Pragma INTERFACE_NAME is required if the Ada identifier differs from the HP C subprogram name.

Note that the parameter in the preceding example may be of mode IN or mode IN OUT. In HP C, user parameters may be passed by value or by reference.

When calling interfaced HP C subprograms, scalar and access parameters of mode IN are passed by value; the value of the parameter object is copied and pushed on the stack. All other types of IN parameters (array, records) and parameters of mode OUT and IN OUT are passed by reference; the address of the parameter object is pushed on the stack.

In general, parameters passed to HP C subprograms from HP Ada programs are passed in 32-bit containers, except for 64-bit real quantities. Real values stored in 32 bits and real values stored in 64 bits are passed intact as parameters between external interfaced subprograms written in HP C and your Ada program. The capability to automatically extend 32-bit real types (Ada FLOAT) to 64-bit values when passed as parameters is not supported by the HP Ada compiler.

Only scalar types (integer, floating point, character, boolean, and enumeration types) and access types are allowed for the result returned by an external interfaced function subprogram written in HP C.

When binding and linking Ada programs with interfaced subprograms written in HP C, the libraries libc.a and libm.a are usually required. The mother program ada(1) will automatically provide the -lc -lm directives to the linker. You are not required to specify "-lc -lm" when binding the Ada program on the ada command line.

For more information about C Language interfacing, see the following manuals: *HP-UX Concepts and Tutorials: Programming Environment*, the *HP-UX Concepts and Tutorials: Device I/O and User Interfacing*, and the *HP-UX Portability Guide*. For more general information about passing Ada types, see Section 14.1.

F 14.3.1 Scalar Types and HP C Subprograms

Scalar types are a class of Ada types that includes integer, real, and enumeration types. Character and boolean types are also scalar types because they are predefined enumeration types. This section discusses passing scalar types between Ada programs and subprograms written in HP C.

Scalar type parameters of mode IN are passed by value. Scalar type parameters of mode OUT or IN OUT are passed by reference.

F 14.3.1.1 Integer Types and HP C Subprograms

When passed by value to an HP C subprogram, all integer values are extended to 32 bits to conform to the HP C parameter passing conventions. HP Ada INTEGER (32 bits) types passed by value are pushed onto the stack without alteration. Values of type SHORT_SHORT_INTEGER (8 bits) and type SHORT_INTEGER (16 bits) are sign extended (on the left) to 32 bits and pushed on the stack.

When passed by reference, the integer values are not changed; the original values are not sign extended to 32 bits. The 32-bit address pointer to the integer object is pushed on the stack.

When passing integers by reference, note that an Ada SHORT_SHORT_INTEGER (8 bits) has no equivalent integer representation in HP C.

Table F-15 summarizes the integer correspondence between Ada and C.

Table F-15. HP Ada/HP C Integer Correspondence

Ada	HP C	Bit Length
SHORT_SHORT_INTEGER	no equivalent representation	8
SHORT_INTEGER	short and short int	16
INTEGER	int, long, and long int	32

If a SHORT_SHORT_INTEGER is passed by reference from HP Ada to HP C, the external subprogram must treat the reference as being a pointer to char, and not as a pointer to short. Otherwise an Ada SHORT_SHORT_INTEGER cannot be meaningfully accessed by HP C subprograms as any type of integer. Because HP C permits integer operations on objects of type char, an Ada SHORT_SHORT_INTEGER can be used and modified (as an object of type char) in an interfaced subprogram.

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP C if care is taken with respect to differences in the interpretation of 8-bit quantities.

F 14.3.1.2 Enumeration Types and HP C Subprograms

When HP Ada passes Ada enumeration types by value to an HP C subprogram, the enumeration types are treated as the underlying integer value. That value is either an 8-bit quantity or a 16-bit quantity (see Section F 12) and is passed according to the convention given previously for integer values. When passed by value, the integer is copied to the stack and is automatically extended to 32 bits.

When passed by reference, the value is not altered by sign extension and a 32-bit address pointer to the enumeration object is pushed on the stack.

Enumeration types in HP C are represented in the same way as in Ada, except that the values are always expressed as 16-bit integers (no matter how many elements there are). When HP C passes enumeration types as *value* parameters, the values are extended to 32 bits, with the high order 16 bits disregarded. Ada's automatic extension to 32 bits means that Ada enumeration type values are in the correct form for HP C subprograms whether stored in 8 or 16 bits.

When passed by *reference*, the original values are not extended. Therefore Ada's 8-bit enumeration objects have no representation in HP C and cannot be used directly by an HP C subprogram. However, an Ada 8-bit enumeration value passed by reference to HP C *can* be accessed as a character in C (using pointer-to-char). Because HP C permits integer operations on objects of type char, an Ada 8-bit enumeration value passed by reference to HP C can be successfully used and modified by HP C (as an object of type char). Enumeration types are allowed for the result returned by an external interfaced subprogram written in HP C.

F 14.3.1.3 Boolean Types and HP C Subprograms

The type boolean is not defined in HP C, and the HP Ada representation of booleans does not correspond to any type in HP C. Ada booleans can be converted to integers and then passed to external interfaced subprograms written in HP C.

In Ada, BOOLEAN'POS(TRUE) has a result of type UNIVERSAL_INTEGER with a value of one and BOOLEAN'POS(FALSE) has a result of type UNIVERSAL_INTEGER with a value of zero. Thus the Ada attribute 'POS can be used to convert an Ada boolean type to an integer type. The integer type can then be passed to an HP C external subprogram.

Boolean types are represented internally as an 8-bit enumeration type of the predefined type (FALSE, TRUE). Although a predefined enumeration type, Ada boolean values are represented in 8 bits in a manner different from other enumeration types in the HP implementation of Ada . The internal representation of FALSE corresponds to 2#0000_0000# and TRUE corresponds to 2#1111_1111# (that is, all zeros or all ones in 8 bits). Note that the value of BOOLEAN'POS(TRUE) is 1 and the value of BOOLEAN'POS(FALSE) is 0. Thus Ada booleans could also be passed as enumeration types in the same way as they are passed as integer types. To ensure portability, however, passing the 'POS of a boolean type as an integer is recommended.

Boolean types are allowed for the result returned by an external interfaced subprogram written in HP C, when care is taken to observe the internal representation.

F 14.3.1.4 Character Types and HP C Subprograms

Ada character types (including the predefined type CHARACTER) correspond exactly with the type char in HP C. Both the Ada and HP C types have the same internal representation and size.

Character types are represented as 8 bits in the HP implementation of Ada. When passed by value to an interfaced subprogram written in HP C, values of type character are zero extended on the left to 32 bits and pushed on the stack; the actual value is the low-order (rightmost) 8 bits.

When passed by reference, the character values are not altered and a 32-bit address of the character object is pushed on the stack.

Character values, whether passed by value or reference to HP C, can be used directly by the interfaced subprogram as values of type char (using pointer-to-char when passed by reference).

Character types are allowed for the result returned by an external interfaced subprogram written in HP C.

F 14.3.1.5 Real Types and HP C Subprograms

This section discusses passing fixed point types and floating point types to HP C.

Fixed Point Real Types

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types cannot be returned as function results from interfaced subprograms written in HP C. For information on the Ada type STRING, see Section F 14.3.3 because Ada strings are a special case of arrays of characters.

Floating Point Real Types

Both Ada and HP C use the IEEE (draft 10.0, P754) floating point conventions for both 32-bit and 64-bit floating point real numbers. The HP C type float and the Ada type FLOAT are implemented on 32 bits and have both the representation and length of the IEEE standard 32-bit real. The HP C type double and the Ada type LONG_FLOAT are implemented on 64 bits. Both have the representation and length of the IEEE standard 64-bit real.

HP C parameter passing conventions require that all 32-bit (that is, C type float) values be extended to 64 bits (C type double) when they are passed as parameters or returned as results. The HP Ada compiler supports this convention for Ada type LONG_FLOAT, but does *not* follow the convention for Ada type FLOAT. Thus an Ada FLOAT value may *not* be passed by value directly to HP C, nor returned from HP C. An Ada FLOAT value may be passed by reference to HP C, because C does not expect such parameters to be extended. A beneficial consequence of this restriction on 32-bit floating point values is that the 32-bit HP C math library is callable from within Ada programs.

When passed by *value* to interfaced subprograms written in HP C, values of type FLOAT and LONG_FLOAT are copied, and pushed on the stack. Float values stored in 32 bits are pushed on the stack as 32-bit quantities (without automatic extension to 64 bits) and values stored in 64 bits are pushed on the stack as 64-bit quantities.

When passed by *reference* to an interfaced subprogram written in HP C, the original values are not altered and a 32-bit address of the object is pushed on the stack.

The 64-bit floating point type LONG_FLOAT can be returned as a result from an external interfaced function subprogram written in HP C.

The 32-bit floating point type FLOAT *cannot* be directly returned as a result from an external interfaced function subprogram written in HP C. It is possible to pass Ada FLOAT values to HP C by value and to return Ada FLOAT values from HP C indirectly if the C routine uses a union containing a float field and a long int field.

A sample HP C function to illustrate this follows. This example computes the square of an Ada FLOAT value. The parameter in the C function is declared to be a union type that matches the size of the Ada FLOAT being passed (32 bits). The function uses the float field (f) of the union to access the Ada floating value. The function stores the result in a union of the same type. It returns the long int field (i) of the union that overlays (occupies the same storage as) the float field. The long int field is returned in a manner in which Ada expects a FLOAT function result (occupying 32 bits) to be returned. Note that neither the float field of the union nor the entire union should be returned. (The former would be returned incorrectly as a double and the latter as an HP C structured type that would be inconsistent with the expected Ada FLOAT result.)

The following example indirectly returns a float result to Ada.

```
typedef union { float f; long int i} ada_float;

int squareit (parm) ada_float parm;
{
ada_float result;

   result.f = parm.f * parm.f;

   return result.i;
}
```

A sample Ada procedure to call this HP C function is the following:

```
--example to workaround restriction on Ada FLOAT result
with TEXT_IO;
procedure SQUARE_PI is
pi : constant FLOAT :=3.1415926;
pi2 : FLOAT;

package FLOATER is new TEXT_IO.FLOAT_IO(FLOAT);

function SQUARE ( f : FLOAT) return FLOAT;
pragma INTERFACE (C, SQUARE);
pragma INTERFACE_NAME (SQUARE, "squareit");

begin -- SQUARE_PI
   pi2 := SQUARE (pi);
   TEXT_IO.PUT ("Pi squared = ");
   FLOATER.PUT (pi2);
   TEXT_IO.NEW_LINE;
end SQUARE_PI;
```

F 14.3.2 Access Types and HP C Subprograms

Ada access types are pointers to Ada objects. Ada access types to simple objects (scalar type or other access type objects) are actually pointers to the designated object. Ada access type values pointing to composite types (record or array types) are pointers to Ada descriptors for that object rather than pointers to the actual object.

NOTE

> You are cautioned that the descriptors used by HP Ada are not identical to descriptors used by HP C. An Ada access type value that points to a composite type has no meaning when passed as a parameter to a subprogram written in HP C.

Ada access types of mode IN are passed by *value*. When an access type is passed by value, a copy of the 32-bit object address is pushed on the stack. If the type is passed by *reference*, however, a 32-bit address pointer to the object address location is pushed on the stack. This is effectively a double indirect address to the underlying object. (See Figure F-1 in Section 14.1.2).

Ada access types may be returned as function results from external interfaced subprograms written in HP C.

An object designated by an Ada access type can be passed to an HP C external subprogram subject to rules applicable to the type of the underlying object.

F 14.3.3 Array Types and HP C Subprograms

For more information, see the comments on general parameter passing in Section F 14.1.3. Note that Ada arrays with SHORT_SHORT_INTEGER and 8-bit enumeration type components do not correspond directly to any HP C array type. Such arrays can be accessed in C as an array of type char. The subprogram must access and modify such elements in a manner appropriate to the actual Ada type; note that an Ada 8-bit enumeration type has a limited range (0 through 128). Arrays are not supported as function results from external interfaced HP C function subprograms.

HP Ada strings are one special case of arrays that do not end with an ASCII null character (\000), as required of strings in HP C. You must append a null character to the end of an Ada string, if that string is to be sent to an external interfaced HP C subprogram. The excess null terminator character may need to be removed from strings passed from HP C to Ada. Caution is required regarding the assignment of unconstrained strings into array objects in Ada. In particular, there is a restriction that an interfaced external subprogram cannot directly return an unconstrained array (such as a string).

The examples on the next two pages illustrate the handling of strings in C and in Ada. In the first example, an Ada string is passed to C. Note the need to explicitly attach a null character so that the C routine can locate the end of the string.

The HP C routine:

```
--passing an Ada string to a C routine
void read_ada_str (var_str)
   char *var_str;
{
   printf ("C: Received value was : %s \n", var_str);
}
```

The Ada routine:

```
Procedure send_ada_str is

   -- Declare an interfaced procedure that sends an
   -- Ada-String to a C-subprogram
   procedure READ_ADA_STR ( VAR_STR : STRING);
   pragma INTERFACE (C, READ_ADA_STR);

begin -- send_ada_str

   -- Test the passing of an Ada string to a C routine
   READ_ADA_STR ( "Ada test string sent to C " & ASCII.NUL);

end send_ada_str ;
```

In the second example, a C string is passed to Ada. Note the need for Ada to determine the length of the string parameter explicitly by using a brute force loop that reads, counts, and compares each character while looking for the end of the C string. The length is required because Ada needs a string of the correct length into which it can assign the characters. Fortunately, Ada supports the allocation of such a string (of arbitrary but fixed length) at run time.

The HP C routine:

```
-- Reading a variable length string from a C function
       char *read_c_str()
       {
          char *local_string;

          local_string = "a C string for Ada.";
          return local_string;
       }
```

The Ada routine:

```
with text_io;
Procedure read_cstring is
   -- Maximum size string expected from C func.
   MAX_STR_SIZE : constant := 255 ;

   subtype MAX_STR is STRING ( 1 .. MAX_STR_SIZE );
   type ACC_STR is access MAX_STR;
   READ_STR : ACC_STR;

   -- Declare an interfaced procedure that returns a pointer
   -- to a C string (actually a pointer to a character)
   function READ_C_STR return ACC_STR;
   pragma INTERFACE (C, READ_C_STR);

   C_STRING_LENGTH : NATURAL := 0 ;  --set length
                                     --initially to zero
begin -- read_cstring
   -- Read an unconstrained string from a C function that returns
   -- a variable length, null terminated string of characters.

   -- assign the pointer to the C string to a statically sized
   -- Ada string access,  so both point to same storage.
   READ_STR := READ_C_STR ;

   -- calculate the length of the C string excluding null terminator
   while (READ_STR.all ( C_STRING_LENGTH + 1) /= ASCII.NUL) loop
   C_STRING_LENGTH := C_STRING_LENGTH + 1;
   end loop;

   declare -- block
   -- allocate an Ada string of exactly the right length
   -- to hold the C string (excluding the null terminator)
   ADA_STRING : STRING ( 1.. C_STRING_LENGTH);

   begin -- block
   -- copy the string from C storage to Ada storage
   ADA_STRING :=  READ_STR.all ( 1 .. C_STRING_LENGTH );

   -- print the string
   TEXT_IO.PUT_LINE ("Ada read: " & ADA_STRING );

   end; -- block

end read_cstring ;
```

F 14.3.4 Record Types and HP C Subprograms

For more information, see comments on general parameter passing in Section F 14.1.4. Ada records may be passed as parameters to external interfaced subprograms written in HP C if care is taken regarding the record layout and access to record discriminant values. See Section F 12 for more information on record type layout.

Record types can*not* be returned as function results from external interfaced subprograms written in HP C.

F 14.3.5 Task Types and HP C Subprograms

Task types cannot be passed as procedure or function parameters in HP Ada. Task types cannot be returned as function results from external interfaced subprograms written in HP C.

F 14.3.6 Private Types and HP C Subprograms

Private types cannot be passed as parameters in HP Ada. Private types cannot be returned as function results from external interfaced subprograms written in HP C.

F 14.4 Calling HP FORTRAN 77 Language Subprograms

When calling interfaced HP FORTRAN 77 subprograms, the following form is used:

```
pragma INTERFACE(FORTRAN, Ada_subprogram_name)
```

This form is used to identify the need for HP FORTRAN 77 parameter passing conventions.

To call the HP FORTRAN 77 subroutine

```
Subroutine FSUB (Parm)
Integer*4 Parm
. . .
end
```

you need this interfaced subprogram declaration in Ada:

```
procedure FSUB (PARM1 : in out INTEGER);
pragma INTERFACE (FORTRAN, FSUB);
```

The external name specified in the Ada interface declaration can be any Ada identifier. If the Ada identifier differs from the FORTRAN 77 subprogram name, pragma INTERFACE_NAME is required. No special handling of leading underscores is required as this is handled by the compiler to conform to standard calling conventions. The pragma ensures that the underscore required in front of the HP FORTRAN 77 subroutine name is correctly inserted by the compiler.

Note that the parameter in the example above must be of mode IN OUT. In HP FORTRAN 77, all user parameters must be passed by reference. The HP FORTRAN 77 compiler creates some implicit parameters that are passed by value not by reference.

Only scalar types (integer, floating point, boolean, and character types) are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77. Boolean and access type results are not supported.

The FORTRAN libraries libI77.a and libF77.a are required when linking Ada programs that contain external interfaced subprograms written in FORTRAN on the HP 9000 Series 300 implementation of Ada. This may be done by including the following options in the ada(1) command line: "-lI77 -lF77". Note that, unlike the optional requirement for HP C interfaced subprograms, the HP FORTRAN 77 options must be inserted by the user. The ada(1) mother program does not automatically search the HP FORTRAN 77 libraries as the program searches the HP C libraries.

For more information, see the following manuals:

- *FORTRAN/9000 Reference*
- *FORTRAN77 Programming with HP Computers*
- *HP-UX Portability Guide*

For general information about passing types to interfaced subprograms, see Section 14.1.

F 14.4.1 Scalar Types and HP FORTRAN 77 Subprograms

This section describes considerations when passing scalar types between Ada programs and subprograms written in HP FORTRAN 77. Scalar types are a class of Ada types that includes integer, real, and enumeration types. Because character and boolean types are predefined enumeration types, they are also scalar types.

All Ada scalar type parameters are passed by reference to FORTRAN external subprograms. Scalar type parameters therefore must be declared as mode IN OUT to be passed by reference.

F 14.4.1.1 Integer Types and HP FORTRAN 77 Subprograms

When passing parameters to an HP FORTRAN 77 language subprogram, all scalar parameters (which includes integer parameters) must be declared as IN OUT parameters in the Ada program to conform to the HP FORTRAN 77 parameter passing conventions. Then, the parameters will be passed by reference. A 32-bit address pointer to the integer object is pushed on the stack.

Table F-16 summarizes the correspondence between integer types in HP Ada and HP FORTRAN 77.

Table F-16. HP Ada/HP FORTRAN 77 Integer Correspondence

Ada	HP FORTRAN 77	Bit Length
SHORT_SHORT_INTEGER	no equivalent representation	8
SHORT_INTEGER	INTEGER*2	16
INTEGER	INTEGER*4	32

An Ada SHORT_SHORT_INTEGER (stored in 8 bits) has no equivalent integer representation in HP FORTRAN 77, and is therefore incompatible with any HP FORTRAN 77 integer value. Ada values of type SHORT_SHORT_INTEGER must be converted to a longer integer type before being passed by reference as a parameter to an HP FORTRAN 77 subroutine.

The compatible types are the same for procedures and functions. Compatible Ada integer types are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77.

Ada semantics do not allow parameters of mode IN OUT to be passed to function subprograms. Therefore for Ada to call HP FORTRAN 77 external interfaced function subprograms, each scalar parameter's address must be passed. The use of the supplied package SYSTEM facilitates this passing of the object's address. The parameters in an HP FORTRAN 77 external function must be declared as in the ex mple on the following page:

```
-- Ada declaration
with SYSTEM;
VAL1   : INTEGER;  -- a scalar type
VAL2   : FLOAT  ;  -- a scalar type
RESULT : INTEGER;
function FTNFUNC ( PARM1, PARM2 : SYSTEM.ADDRESS) return INTEGER;
```

The external function must be called from within Ada as follows:

```
RESULT := FTNFUNC (VAL1'ADDRESS, VAL2'ADDRESS);
```

Because this has the effect of obscuring the types of the actual parameters, it is suggested that such declarations be encapsulated within an INLINED Ada body so that the parameter types are made visible. An example follows:

```
-- specification of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT    ) return INTEGER;
pragma INLINE (FTNFUNC);

with SYSTEM;
-- body of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT    ) return INTEGER is
   function FORTFUNC (P1, P2 : SYSTEM.ADDRESS) return INTEGER;
   pragma INTERFACE (FORTRAN, FORTFUNC);

begin -- function FTNFUNC
      return FORTFUNC (PARM1'ADDRESS, PARM2'ADDRESS);
end FTNFUNC;
```

In the previous example, the name of the interfaced external function subprogram (written in HP FORTRAN 77) is FORTFUNC. This name is declared in the following way:

```
Integer*4 FUNCTION  FORTFUNC (I, X)
Integer*4 I
Real*4 X
   ...
end
```

F 14.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms

The HP FORTRAN 77 language does not support enumeration types. However, objects that are elements of Ada's enumeration type can be passed as INTEGER as the underlying integer representation of the element. Then, those objects can be used in an HP FORTRAN 77 routine as INTEGER, similar to the way Ada booleans are passed (for more information, see the following section).

F 14.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms

In HP Ada, the type BOOLEAN is represented in 8 bits and cannot be used as a logical or short logical (that is, LOGICAL*2) in HP FORTRAN 77. However, an INTEGER may be passed instead of the boolean value (which can be mapped to a LOGICAL*2).

Although a predefined enumeration type, Ada boolean values are represented in 8 bits in a manner different from other enumeration types in the HP implementation of Ada. The internal representation of FALSE corresponds to 2#0000_0000# and TRUE corresponds to 2#1111_1111# (that is, all zeros or all ones in 8 bits). Note that the value of BOOLEAN'POS(TRUE) is 1 and the value of BOOLEAN'POS(FALSE) is 0.

The Ada attribute 'POS can be used to convert an Ada boolean to an integer type, which can then be passed to an HP FORTRAN 77 external subprogram. Note that in Ada, BOOLEAN'POS(TRUE) has a result of (UNIVERSAL_INTEGER) one, and that BOOLEAN'POS(FALSE) has a result of (UNIVERSAL_INTEGER) zero. This result can then be converted to an integer type of 16 bits and subsequently can be treated as a LOGICAL*2 in HP FORTRAN 77 and passed as a parameter.

Likewise, function results of type LOGICAL from an HP FORTRAN 77 function subprogram may be considered in Ada as having a SHORT_INTEGER type representing the position (attribute 'POS) of the equivalent boolean value.

For example, an HP FORTRAN 77 subroutine of the form

```
SUBROUTINE LOGICAL_SUB (LOGICAL_PARM)
LOGICAL*2 LOGICAL_PARM
...
RETURN
```

can be called after suitable declarations in Ada as follows:

```
subtype FORTRAN_LOGICAL is SHORT_INTEGER;  -- 16 bits
BVAR  : BOOLEAN;  -- will be input boolean value

-- define a place holder for equivalent integer
BTEMP : SHORT_INTEGER;  -- integer on 16 bits

procedure BOOL_PROC(B_PARM : in out FORTRAN_LOGICAL);
pragma INTERFACE (FORTRAN, BOOL_PROC);
pragma INTERFACE_NAME ( BOOL_PROC, LOGICAL_SUB);
```

The above subroutine is called from the Ada program as:

```
BTEMP := FORTRAN_LOGICAL(BOOLEAN'POS(B_VAR)); -- get equivalent
BOOL_PROC( BTEMP );  -- the call to the external
                     -- subroutine LOGICAL_SUB
```

Note that the expression

```
BOOLEAN'POS(B_VAR)
```

evaluates to a UNIVERSAL_INTEGER value of one if B_VAR is TRUE or a UNIVERSAL_INTEGER value of zero if B_VAR is FALSE, while the expression of the form

```
BTEMP := FORTRAN_LOGICAL(...);
```

is an explicit type conversion (on the right) and assignment to a 16-bit integer type (on the left) that is required to be passed to map into an equivalent sized HP FORTRAN 77 LOGICAL*2 object.

Note that the call from the Ada program in the preceding example involved an assignment rather than an embedded expression. An Ada restriction *is that an IN OUT parameter must not* be an expression. So, the following example is not legal Ada:

```
BOOL_PROC(FORTRAN_LOGICAL(BOOLEAN'POS(B_VAR))); -- not legal
```

F 14.4.1.4 Real Types and HP FORTRAN 77 Subprograms

This section discusses passing fixed and floating point types to subprograms written in FORTRAN.

Fixed Point Real Types

Ada fixed point types are not supported as parameters or as results of external interfaced subprograms written in HP FORTRAN 77. Ada fixed point types cannot be returned as function results from external interfaced subprograms written in HP FORTRAN 77.

Floating Point Real Types

Parameters of type FLOAT in Ada correspond to the default REAL (REAL*4) format in HP FORTRAN 77. The Ada type LONG_FLOAT is equivalent to the HP FORTRAN 77 type DOUBLE PRECISION (or REAL*8). HP FORTRAN 77 follows the IEEE P754 floating point conventions for both 32-bit and 64-bit floating point numbers. Both 32-bit real values and 64-bit real values can be passed as parameters.

A REAL value from an HP FORTRAN 77 external function subprogram may be returned as a function result of type FLOAT in an Ada program.

A DOUBLE PRECISION (or REAL*8) value from an HP FORTRAN 77 external function subprogram may be returned as a function result of type LONG_FLOAT in an Ada program.

When passed as parameters to an interfaced subprogram written in HP FORTRAN 77, the original real values are not altered and a 32-bit address of the real object is pushed on the stack.

F 14.4.2 Access Types and HP FORTRAN 77 Subprograms

Ada access types have no meaning in HP FORTRAN 77 subprograms because the types are address pointers to Ada objects. The implementation value of an Ada parameter of type ACCESS may be passed to an HP FORTRAN 77 procedure. The parameter in HP FORTRAN 77 is seen as INTEGER*4. The object pointed to by the access parameter has no significance in HP FORTRAN 77; its value would be useful only for comparison operations to other access values.

Parameters or function results of type ACCESS from external interfaced subprograms written in HP FORTRAN 77 are not supported. HP FORTRAN 77 could return an INTEGER and the Ada program can use as the returned value type INTEGER (a matching size, because HP Ada's INTEGER is a 32-bit quantity in this implementation). The significance and use in HP FORTRAN 77 of such a value must be understood as meaningless, as mentioned previously.

F 14.4.3 Array Types and HP FORTRAN 77 Subprograms

For more information, see the general comments on parameter passing of arrays in Section F 14.1.3.

Arrays whose components have an HP FORTRAN 77 representation can be passed as parameters between Ada and interfaced external HP FORTRAN 77 subprograms. For example, Ada arrays whose components are of types INTEGER, SHORT_INTEGER, FLOAT, LONG_FLOAT, or CHARACTER may be passed as parameters. Arrays are not supported as function results from external HP FORTRAN 77 function subprograms.

CAUTION

> Arrays with multiple dimensions are implemented differently in Ada and HP FORTRAN 77. To obtain the same layout of components in memory as a given HP FORTRAN 77 array, the Ada equivalent must be declared and used with the dimensions in reverse order.

Consider the components of a 2-row by 3-column matrix, declared in HP FORTRAN 77 as:

```
INTEGER*4  A(2,3) OR INTEGER*4  A(1:2,1:3)
```

This array would be stored by HP FORTRAN 77 in the following order:

```
A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3)
```

This is referred to as storing in *column major order*; that is, the first subscript varies most rapidly, the second varies next most rapidly, and so forth, and the last varies least rapidly.

Consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..2, 1..3) of INTEGER;
```

This array would be stored by Ada in the following order:

```
A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3)
```

This is referred to as storing in *row major order*, that is the last subscript varies most rapidly; the next to last varies next most rapidly, and so forth, while the first varies least rapidly. Clearly the two declarations in the different languages are no' equivalent. Now, consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..3, 1..2) of INTEGER;
```

Note the reversed subscripts compared with the FORTRAN declaration. This array would be stored by Ada in the following order:

```
A(1,1), A(1,2), A(2,1), A(2,2), A(3,1), A(3,2)
```

If the subscripts are reversed, the layout would be

```
A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3)
```

which is identical to the HP FORTRAN 77 layout. Thus, either of the language declarations could declare its component indices in *reverse* order to be compatible.

To illustrate that equivalent multidimensional arrays require a reversed order of dimensions in the declarations in HP FORTRAN 77 and Ada, consider the following:

The Ada statement

```
FOO : array (1..10,1..5,1..3) of FLOAT;
```

is equivalent to the HP FORTRAN 77 declaration:

```
REAL*4  FOO(3,5,10)
```

Or

```
REAL*4  FOO(1:3,1:5,1:10)
```

Both Ada and HP FORTRAN 77 store a one-dimensional array as a linear list.

F 14.4.4 String Types and HP FORTRAN 77 Subprograms

When a string item is passed as an argument to an HP FORTRAN 77 subroutine from within HP FORTRAN 77, extra information is transmitted in hidden (implicit) parameters. The calling sequence includes a hidden parameter (for each string) which is the actual length of the ASCII character sequence. This implicit parameter is passed in addition to the address of the ASCII character string. The hidden parameter is passed by value, not by reference.

These conventions are different from those of Ada. For an Ada program to call an external interfaced subprogram written in HP FORTRAN 77 with a string type parameter, you must explicitly pass the length of the string object. The length must be declared as an Ada 32-bit integer parameter of mode IN.

The following example illustrates the declarations needed to call an external subroutine having a parameter profile of two strings and one floating point variable.

```
procedure FTNSTR is
   SA: STRING(1..6):= "ABCDEF";
   SB: STRING(1..2):= "GH";
   FLOAT_VAL: FLOAT:=  1.5;
   LENGTH_SA, LENGTH_SB : INTEGER;

   procedure FEXSTR ( S1  : STRING;
                      F   : in out FLOAT;  -- must be IN OUT
                      S2  : STRING; -- strings are passed by reference
                      LS2 : in SHORT_INTEGER; -- len of string S2,
                                              -- must be IN
                      LS1 : in SHORT_INTEGER ); -- len of string S1,
                                                -- must be IN
   pragma INTERFACE (FORTRAN, FEXSTR);

begin -- procedure FTNSTR
  LENGTH_SA := SA'LENGTH;
  LENGTH_SB := SB'LENGTH;
  FEXSTR(SA, FLOAT_VAL, SB, LENGTH_SA, LENGTH_SB);
end FTNSTR;
```

NOTE

Note that the order of the string lengths is in the same order of their appearance in the corresponding parameter and that they appear after all other parameters (at the end of the parameter list). The string lengths must be passed by value, *not* by reference.

The HP FORTRAN 77 external subprogram is the following:

```
SUBROUTINE FEXTR (S1, r, S2)
CHARACTER *(*) S1, S2
REAL*4  r
  ...
END
```

NOTE

HP Ada does *not* allow a string type (constrained or *not*) to be returned from a function interfaced with HP FORTRAN 77. Thus, it is *not* possible to declare an Ada external function that returns a result of type STRING (type STRING is an object of type CHARACTER *(*) in HP FORTRAN 77).

F 14.4.5 Record Types and HP FORTRAN 77 Subprograms

There are no record types in HP FORTRAN 77. Record types *cannot* be passed as parameters in HP FORTRAN 77. Function results of type RECORD from external interfaced subprograms written in HP FORTRAN 77 are *not* supported.

F 14.4.6 Task Types and HP FORTRAN 77 Subprograms

Task types *cannot* be passed as procedure or function parameters in HP Ada. Function results of type TASK from external interfaced subprograms written in HP FORTRAN 77 are *not* supported.

F 14.4.7 Private Types and HP FORTRAN 77 Subprograms

Private types *cannot* be passed as parameters in HP Ada. Function results of type PRIVATE from external interfaced subprograms written in HP FORTRAN 77 are *not* supported.

F 14.4.8 Other FORTRAN Types

The HP FORTRAN 77 types COMPLEX, COMPLEX*8, DOUBLE COMPLEX, and COMPLEX*16 have no direct counterparts in Ada. However, it is possible to declare equivalent types using either an Ada array or an Ada record type. For example, with type COMPLEX in HP FORTRAN 77, a simple Ada equivalent is a user-defined record:

```
type COMPLEX is record
   Real : FLOAT := 0.0;
   Imag : FLOAT := 0.0;
end record;
```

Similarly, an HP FORTRAN 77 double complex number could be represented with the two record components declared as Ada type LONG_FLOAT.

While it is *not* possible to declare an Ada external function that returns such a complex type, an Ada procedure *can* be declared whose first OUT parameter is of type COMPLEX. The Ada procedure would have the same functionality as the HP FORTRAN 77 function, with a different mechanism for returning the result.

F 14.5 Calling HP Pascal Language Subprograms

Calling external interfaced Pascal language subprograms from Ada programs requires the use of the assembly language routines asm_initproc and asm_wrapup. Calls to these procedures must bracket the program containing calls to Pascal routines, but only one call to each of these routines is required per program.

The first of these calls must be made to set up Pascal file system, the Pascal heap manager, and error recovery (for the Pascal procedures), and the second to cleanup from the heap, the file system and error recovery usage that was set up by the previous call. Without placing these calls as required the program behavior may be erroneous or unpredictable. For more information on Pascal interfacing, see the *HP Pascal Language Reference Manual*, and the section, "HP-UX Implementation, Pascal and other Languages". Additional information is available in the *HP-UX Portability Guide*.

Another unusual aspect of calling Pascal subprograms (functions and procedures) is that these routines cannot be compiled separately in Pascal unless enclosed in a module declaration. The consequence of this is that the name of the module is prepended to the procedure or function name with an underscore character separating the two identifiers. This is done by the Pascal compiler, and needs to be taken into account in the Ada calling program in the pragma INTERFACE declaration.

For an external interfaced Pascal subroutine:

```
    module modp;
    export
      procedure subr (var parm : integer);

    implement
      procedure subr (var parm : integer);
      begin
        . . .
    end ;
 end.
```

Note the name of the Pascal external subprogram as viewed externally (from Ada) includes module name modp and the procedure name subr formed into the single identifier modp_subr. We would require the following interfaced subprogram declarations in Ada:

```
type  echo_mode is access INTEGER;

procedure MODP_SUBR (Parm1 : in INTEGER);
pragma INTERFACE (Pascal, MODP_SUBR);

procedure  asm_initproc (echo : echo_mode );
pragma INTERFACE(ASSEMBLER, asm_initproc);

procedure  asm_wrapup;
pragma INTERFACE(ASSEMBLER, asm_wrapup);
```

to call the external Pascal subprogram :

```
aparm  : INTEGER := 0;
echo   : echo_mode := new INTEGER'(0);
noecho : echo_mode := new INTEGER'(1);

asm_initproc(echo);  -- initialize heap, etc

...

MODP_SUBR (aparm); -- the call to Pascal procedure

...

asm_wrapup;      -- cleanup
```

For Pascal, scalar and access parameters of mode IN are passed by value: the value of the parameter object is copied and pushed on the stack. All other types of IN parameters (arrays, records), and parameters of mode OUT and IN OUT are passed by reference: The address of the object is pushed on the stack. This means that in general Ada IN parameters correspond to Pascal value parameters while Pascal VAR parameters correspond to the Ada parameters of either mode IN OUT or mode OUT.

For Pascal external interfaced subprograms called from an Ada program, all parameters are passed in containers of size 32-bits, except for Ada LONG_FLOAT parameters which are passed in 64-bits.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed for the result returned by an external interfaced Pascal function subprograms.

For general information about passing parameters to interfaced subprograms, see Section 14.1.

F 14.5.1 Scalar Types and HP Pascal Subprograms

This section discusses passing scalar types between Ada programs and subprogram written in HP Pascal. Scalar types include integer, real, and enumeration types. Because character and boolean types are predefined enumeration types, they are also scalar types.

Ada scalar type parameters of mode IN are passed by *value*. Scalar type parameters of mode OUT or IN OUT are passed by *reference*.

F 14.5.1.1 Integer Types and HP Pascal Subprograms

Integer types are compatible between Ada and HP Pascal provided their ranges of values are identical. Table F-17 shows corresponding integer types in Ada and HP Pascal.

Table F-17. Ada/HP Pascal Integer Correspondence

Ada	HP Pascal	Bit Length
predefined INTEGER	predefined INTEGER	32
predefined SHORT_INTEGER	user-defined types type I16=-32768...32767;	16
predefined SHORT_SHORT_INTEGER	user-defined type, type I8= -128...127;	8 (Ada) 16 (Pascal)

When passed by *value* to an HP Pascal language subprogram, all integer values are extended to 32 bits to conform to the HP Pascal parameter passing conventions. Ada INTEGER (32 bits) types passed by value are pushed onto the stack without alteration. Values of type SHORT_SHORT_INTEGER (8 bits) and type SHORT_INTEGER (16 bits) are sign extended to 32 bits and pushed on the stack.

When passed by *reference*, the integer values are not changed; the original values are not sign extended to 32 bits. A 32-bit address pointer to the integer object is pushed on the stack.

When passing integers by reference, note that the predefined Ada SHORT_INTEGER (stored in 16 bits) and the Ada SHORT_SHORT_INTEGER (stored in 8 bits) have no equivalent predefined integer representations in HP Pascal. However, an HP Pascal integer can be declared whose range matches the range of these types. Note that a difference in data layout between the two languages for the range -128 through 127 places a restriction on passing a SHORT_SHORT_INTEGER to Pascal. For example, the range and storage size of Ada's SHORT_INTEGER corresponds to the range and storage size of HP Pascal declaration

```
type I16 = -32768 .. 32767 ;
```

The range of Ada's SHORT_SHORT_INTEGER corresponds to the range of HP Pascal declaration

```
type I8 = -128  ..  127 ;
```

However, the storage size is 16 bits in HP Pascal while it is 8 bits in Ada.

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP Pascal, if care is taken with respect to ranges defined for integer quantities. The automatic sign extension helps in passing dissimilar-sized values that are within an acceptable range.

F 14.5.1.2 Enumeration Types and HP Pascal Subprograms

Ada and HP Pascal have different implementations of enumeration types. An Ada enumeration type when passed as a parameter is treated as if it were the underlying integer representation. That value is either an 8-bit quantity (for enumeration sizes <= 128) or a 16-bit quantity (for enumeration sizes > 128). It is passed according to the convention given above for integer values. When passed by *value*, the integer is copied to the stack and automatically extended to 32 bits. In HP Pascal, all enumeration types are 16-bit quantities, regardless of enumeration size. If care is taken with respect to the underlying representation, the automatic extension to 32 bits allows the two implementations to treat enumeration types as equivalent.

When passed by *reference*, the original value is not altered by sign extension and a 32-bit address pointer to the enumeration object is pushed on the stack.

Ada supports the return of a function result that is an enumeration type from an external interfaced function subprogram written in HP Pascal.

F 14.5.1.3 Boolean Types and HP Pascal Subprograms

Although a predefined enumeration type, Ada boolean values are represented in 8 bits in a manner different from other enumeration types in the HP implementation of Ada . The internal representation of FALSE corresponds to 2#0000_0000# and TRUE corresponds to 2#1111_1111# (that is, all zeros or all ones in 8 bits). Note that the value of BOOLEAN'POS(TRUE) is 1 and the value of BOOLEAN'POS(FALSE) is 0. Since HP Pascal treats boolean values differently, an Ada boolean value may not be treated as a Pascal boolean value in an external subprogram, whether passed by value or passed by reference. The Ada attribute POS can be used to convert an Ada boolean type to an integer type.

There is no support for the return of a function result that is a boolean type from an external interfaced function subprogram written in HP Pascal.

F 14.5.1.4 Character Types and HP Pascal Subprograms

Values of the Ada predefined character type may be treated as the type char in HP Pascal external interfaced subprograms. Both the Ada and HP Pascal types occupy 8 bits and have the same internal representation.

Character types are represented as 8 bits in the HP implementation of Ada. When passed by *value* to an interfaced subprogram written in HP Pascal, values of type CHARACTER are extended to 32 bits and pushed on the stack; the actual value is the low-order (rightmost) 8 bit.

When passed by *reference* to an interfaced subprogram written in HP Pascal, the character values are not altered and a 32-bit address of the character object is pushed on the stack.

Ada character values, whether passed by value or reference to HP Pascal, can be used directly by the interfaced subprogram as values of type char.

Ada supports the return of a function result that is a character type from an external interfaced function subprogram written in HP Pascal. The HP Pascal result is of type char while the Ada result is of type CHARACTER.

F 14.5.1.5 Real Types and HP Pascal Subprograms

The following subsections discuss passing Ada real types to interfaced HP Pascal subprograms.

Fixed Point Types

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types cannot be returned as function results from interfaced subprograms written in HP Pascal.

Floating Point Real Types

HP Pascal uses the IEEE P754 (draft 10.0) floating point conventions for both 32-bit and 64-bit floating point numbers. Ada FLOAT values correspond to HP Pascal REAL values. Ada LONG_FLOAT values correspond to HP Pascal LONGREAL values.

In HP Pascal, 32-bit real values are not automatically extended to 64-bit values when passed as parameters between external interfaced subprograms written in HP Pascal and an Ada program. Both 32-bit and 64-bit real types can be returned as results from an external interfaced function subprogram written in HP Pascal.

When passed by *value* to interfaced subprograms written in HP Pascal, values of type FLOAT and LONG_FLOAT are copied and pushed on the stack. Float values of 32 bits are *not* automatically extended to 64 bits when passed by value.

When passed by *reference* to an interfaced subprogram written in HP Pascal, the original values are not altered and a 32-bit address of the object is pushed on the stack.

F 14.5.2 Access Types and HP Pascal Subprograms

Ada access values can be treated as pointer values in HP Pascal external interfaced subprograms because they are address pointers to Ada objects. The Ada allocation and the HP Pascal allocation are completely separate. There must be no explicit reallocation in one language of an object allocated in the other language.

An object designated by an Ada access type can be passed to a HP Pascal external subprogram, subject to rules applicable to the type of the underlying object.

Ada access types of mode IN are passed by value. When an access type is passed by value, a copy of the 32-bit object address is pushed on the stack. If the type is passed by reference, however, a 32-bit address pointer to the object address location is pushed on the stack. This is effectively a double indirect address to the underlying object. (See Figure F-1 in Section F 14.1.2.)

Ada access types may be returned as function results from external interfaced subprograms written in HP Pascal.

F 14.5.3 Array Types and HP Pascal Subprograms

Arrays with components with the same representation have the same representation in Ada and HP Pascal.

Arrays cannot be passed by value from Ada to HP Pascal. Only the passing of arrays declared as a VAR parameter in an HP Pascal subprogram to an Ada program is supported.

Array types cannot be returned as function results from external interfaced subprograms written in HP Pascal.

Note that the function results of array types are not supported between HP Pascal and Ada.

F 14.5.4 String Types and HP Pascal Subprograms

Passing variable length strings between Ada and HP Pascal is supported with some restrictions. The parameters must be passed by reference only. HP Pascal programs must declare VAR parameters and the Ada program must declare the parameters to be of mode IN OUT or OUT to ensure passing by reference.

Although there is a difference in the implementation of the type STRING in the two languages, with suitable declarations you can create compatible types to allow the passing of both Ada strings and HP Pascal strings. An Ada string corresponds to a packed array of characters in Pascal. The following example illustrates the declaration of compatible types for passing an Ada string between an Ada program and an HP Pascal subprogram.

HP Pascal subprogram:

```
-- passing an Ada STRING type to an HP Pascal routine ---
  module p;
  export
      type string80 = packed array [1..80] of char;
      procedure ex1 ( var s   : string80; len : integer );
   implement
      procedure  ex1;
      begin
         ... (* update/use the Ada string as a PAC *)
      end;
   end.
```

Ada program:

```
-- Ada calling HP Pascal procedure with Ada STRING
   procedure AP_1 is

      -- Define Ada string corresponding to HP Pascal packed array of char
      subtype STRING80 is STRING ( 1..80 );

      -- Ada definition of HP Pascal procedure to be called, with an
      -- Ada STRING parameter, passed by reference.
      procedure  P_EX1 (S     :in out STRING80;
                        LEN   :        integer );
      pragma INTERFACE (PASCAL, P_EX1);

      -- Define the HP Pascal initialization routine interfaces
      procedure Pstart ( argc : INTEGER; argv : STRING );
      pragma INTERFACE (ASSEMBLER, Pstart );
      pragma INTERFACE_NAME ( Pstart, "__Pstart" ); --two underscores
                                                    --are required
      S      : STRING80;

   begin -- AP_1
      Pstart ( 0, "" ); -- initialize HP Pascal environment
      S(1..23) := "Ada to HP Pascal Interface";
      P_EX1 (S, 23);      -- Call the HP Pascal subprogram
   end AP_1;
```

An HP Pascal STRING type corresponds to a record in Ada that contains two fields: an integer field containing the string length, and an Ada STRING field containing the string value. The following example illustrates the declaration of compatible types for passing an HP Pascal string between an Ada program and a Pascal subprogram.

Pascal subprogram:

```
-- passing an HP Pascal STRING type from Ada to an HP Pascal routine
module p;
export
   type string80 = string [80];
   procedure ex2 ( var s : string80 );
implement

   procedure  ex2;
   var
      str : string80 ;
   begin
     ... --update/use the HP Pascal string
   end;
end.
```

Ada program:

```
-- Ada calling HP Pascal procedure with HP Pascal STRING
procedure AP_2 is

   -- Define Ada string corresponding to HP Pascal packed array of char
   type PASCAL_STRING80 is
      record
         LEN : INTEGER;
         S   : STRING ( 1..80 );
      end record;

      -- Ada definition of HP Pascal procedure to be called, with a
      -- HP Pascal STRING parameter, passed by reference.
   procedure  P_EX2 (S:in out PASCAL_STRING80);
   pragma INTERFACE (PASCAL, P_EX2);

   -- Define the HP Pascal initialization routine interfaces
   procedure Pstart ( argc : INTEGER; argv : STRING );
   pragma INTERFACE (ASSEMBLER, Pstart );
   pragma INTERFACE_NAME ( Pstart, "__Pstart" ); -- two underscores

   PS     : PASCAL_STRING80;

begin -- AP_2

   Pstart ( 0, "" ); -- initialize HP Pascal environment
   PS.S(1..23) := "Ada to HP Pascal Interface"; -- assign value field
   PS.LEN := 23;  -- set string length field
   P_EX2 ( PS );  -- call the HP Pascal subprogram

end AP_2;
```

F 14.5.5 Record Types and HP Pascal Subprograms

See the comments on general parameter passing in Section F 14.1.4.

Records cannot be passed by value from Ada to HP Pascal. Only the passing of records declared as VAR parameters in the HP Pascal subprogram to Ada programs is supported. The record is passed by reference.

Record types cannot be returned as function results from external interfaced subprograms written in HP Pascal.

F 14.5.6 Task Types and HP Pascal Subprograms

Task types cannot be passed as parameters in Ada. Task types cannot be returned as function results from external interfaced subprograms written in HP Pascal.

F 14.5.7 Private Types and HP Pascal Subprograms

Private types cannot be passed as parameters in Ada, and private types cannot be returned as function results from external interfaced subprograms written in HP Pascal.

F 14.6 Summary

Table F-18 shows how various Ada types are passed to subprograms.

Table F-18. Modes for Passing Parameters to Interfaced Subprograms

Ada Type	Mode	Passed By
ACCESS, SCALAR -INTEGER -ENUMERATION -BOOLEAN -CHARACTER -REAL	IN	value
ARRAY, RECORD	IN	reference
all types except TASK, PRIVATE, and FIXED POINT REAL	IN OUT	reference
all types except TASK, PRIVATE, and FIXED POINT REAL	OUT	reference
TASK PRIVATE FIXED POINT REAL	N/A	not passed

Table F-19 summarizes general information presented in Section 14.1.

Table F-19. Types Returned as External Function Subprogram Results

Ada Type	HP Assembly Language	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed[1]	allowed
CHARACTER	allowed	allowed	not allowed	allowed
BOOLEAN	allowed	allowed	not allowed[1]	not allowed
REAL FIXED	not allowed	not allowed	not allowed	not allowed
REAL FLOAT	allowed	allowed[2]	allowed	allowed
ACCESS	allowed	allowed	not allowed[1]	allowed
ARRAY	not allowed	not allowed	not allowed	not allowed
RECORD	not allowed	not allowed	not allowed	not allowed
TASK	not allowed	not allowed	not allowed	not allowed
PRIVATE	not allowed	not allowed	not allowed	not allowed

Notes for Table F-19.

1 Pass as an integer equivalent.
2 Some restrictions apply to Ada FLOAT types (in passing to HP C subprograms).

Table F-20 summarizes information presented in Sections F 14.2 through F 14.5.

Table F-20. Parameter Passing in the Series 300 Implementation

Ada Type	HP Assembly Language	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed[1]	allowed
CHARACTER	allowed	allowed	allowed	allowed
BOOLEAN	allowed	allowed	not allowed[1]	allowed[2]
REAL FLOAT	allowed	allowed	allowed	allowed
REAL FIXED	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed	allowed
ARRAY[3]	not allowed	allowed	allowed[4]	allowed
STRING	not allowed	allowed[5]	allowed	not allowed[6]
RECORD	not allowed	allowed	not allowed	allowed
TASK	not allowed	not allowed	not allowed	not allowed
PRIVATE	not allowed	not allowed	not allowed	not allowed

[1] Can be passed as an equivalent integer value.
[2] Passed by value only.
[3] Using only arrays of compatible component types.
[4] See warning on layout of elements in section for each language.
[5] Special handling of null terminator character is required.
[6] Ada strings can be passed to a Pascal PAC (Packed Array of Characters)

F 14.7 Potential Problems Using Interfaced Subprograms

The Ada run-time executive for the HP 9000 Series 300 computer uses signals in a manner that generally does not interfere with interfaced subprograms. However, some HP-UX routines are interruptible by signals. These routines, if called from within interfaced external subprograms, may create problems. You need to be aware of these potential problems when writing external interfaced subprograms in other languages that will be called from within an Ada main subprogram. See `sigvector(2)` in the *HP-UX Reference* for a complete explanation of interruptibility of operating system routines.

The following should be taken into consideration:

- SIGALRM is sent when a DELAY statement is being timed.
- SIGVTALRM is sent when round-robin scheduling is used in tasking programs.
- There are "slow" HP-UX routines (see `sigvector(2)`) that can be interrupted by signals generated by the Ada run-time executive. Therefore you must temporarily block these signals. These "slow" routines include HP-UX I/O calls to interactive devices.
- No received signals will be lost while they are blocked, owing to the use of HP-UX reliable signals.
- Any signals blocked in interfaced code must be unblocked before leaving interfaced code to return to the calling Ada program.

The Ada run-time executive uses two time-related signals for control of task scheduling and the execution of DELAY statements. The executive uses SIGALRM to indicate the end of a DELAY time interval because SIGALRM operates in real time. The run time uses SIGVTALRM to denote the end of a task's time slice amount, because SIGALRM operates in process-local (virtual) time.

These two signals may cause problems in interfaced routines because the signals are asynchronous to the external (interfaced) code. Asynchronous in this context means that the signals could occur at any moment and they are not caused by the code executing at the instant they occur. The signals may interrupt some vulnerable routines such as the HP-UX I/O calls mentioned above.

The Ada run-time executive for the HP 9000 Series 300 implementation generates and catches the following HP-UX signals:

SIGALRM -- used for DELAY statements
SIGVTALRM -- used for task scheduling
SIGSEGV -- STORAGE_ERROR
SIGFPE -- NUMERIC_ERROR, Illegal trap
SIGEMT -- unimplemented instruction
SIGBUS -- PROGRAM_ERROR, User exception, Illegal trap, Erroneous instruction
SIGILL -- CONSTRAINT_ERROR, NUMERIC_ERROR, Illegal trap

On Ada 300, the signals SIGBUS, SIGSEGV, and SIGFPE can result in the predefined Ada exceptions listed above, provided their associated trap number (see `trapno (2)`) indicates a condition defined by these exceptions. You should *not* attempt to modify these trap conditions.

NOTE

The signals SIGALRM and SIGVTALRM are not always generated in tasking programs. SIGALRM is only generated if and when a DELAY statement is encountered. SIGVTALRM is only generated if time slicing has not been turned off with the binder option "-W b,s,0". Non-tasking programs do not use SIGVTALRM because for such programs time slicing is meaningless.

Attempts to trap or ignore any of the signals listed above (except as described subsequently and in section F 9.1) by an interfaced subprogram may have unpredictable results. Subprograms that are part of the Ada run-time executive are the only exception; however, you have access to only a few such programs. See the *Ada User's Guide* for more details.

Problems can arise if an interfaced subprogram initiates a "slow" operating system function that can be interrupted by a signal (for example, a read on a terminal or a wait for a child process to complete). Problems can also arise if an interfaced subprogram can be called by more than one task and is not reentrant. If an Ada reserved signal occurs during such an operation or non-reentrant region, the program may function erroneously.

For example, an Ada program that uses DELAY statements and tasking constructs causes the generation of SIGALRM and SIGVTALRM. If an interfaced subprogram needs to perform a potentially interruptible operating system call, or if it might be called from more than one task and is not reentrant, it can be protected by blocking SIGALRM and SIGVTALRM around the operating system call or non-reentrant region. If a SIGALRM or SIGVTALRM signal signifying either the end of a delay period or the need to reschedule a task is received while it is blocked, the signal is not lost, but rather deferred until it is later unblocked. The consequence of this signal blocking is that Ada task scheduling or DELAY statement execution will be affected for the duration of the signal block.

Here is an example of a protected read(2) in an interfaced subprogram written in the C Language.

```
#include <signal.h>
void interface_rout()
{
   long mask;

...

   /* Add SIGALRM  and SIGVTALRM to list of currently
      blocked signals. (see sigblock(2)).     */

  mask = sigblock (( 1L << (SIGALRM-1)) | (1L << (SIGVTALRM-1)));

  ...   read (...) ;   /* or non reentrant region */

  setsigmask (mask) ;   /* return to previous mask */

}
```

If any Ada reserved signal other than SIGALRM or SIGVTALRM is to be similarly blocked, SIGALRM and SIGVTALRM must be either already blocked or blocked at the same time. When any Ada reserved signal

other than SIGALRM or SIGVTALRM is unblocked, SIGALRM and SIGVTALRM must be unblocked at the same time, or as soon as possible thereafter.

Any Ada reserved signal blocked in interfaced code should be unblocked before leaving that code, or as soon as possible thereafter, to avoid unnecessarily stalling the Ada run-time executive. Failure to follow these guidelines will cause improper delay or tasking operation.

An alternative method of protecting interfaced code from signals is described in the *Ada User's Guide* in the section on "Execution-Time Topics." The two procedures SUSPEND_ADA_TASKING and RESUME_ADA_TASKING from the package UNIX_ENV supplied by HP can be used within an Ada program to surround a critical section of Ada code or a call to external interfaced subprogram code with a critical section.

F 14.8 Input-Output From Interfaced Subprograms

Using I/O from interfaced subprograms written in other languages requires caution. Some areas in which problems can arise are discussed in this section.

F 14.8.1 Files Opened by Ada and Interfaced Subprograms

An interfaced subprogram should *not* attempt to perform I/O operations on files opened by Ada. Your program should not use HP-UX I/O utilities intermixed with Ada I/O routines on the same file. If it is necessary to perform I/O operations in interfaced subprograms using the HP-UX utilities, open and close those files with HP-UX utilities.

F 14.8.2 Preconnected I/O and Interfaced Subprograms

The standard HP-UX files stdin and stdout are preconnected by Ada I/O. If non-blocking interactive I/O is used, additional file descriptors will be used for interactive devices connected to stdin or stdout. Ada does not preconnect stderr, which is used for run-time error messages. For more details, see the section on Ada I/O in the *LRM* and the section on using the Ada Compilation system in the *Ada User's Manual*.

F 14.8.3 Interactive I/O and Interfaced Subprograms

The default I/O system behavior is NON-BLOCKING for Ada programs with tasking and BLOCKING for sequential (non-tasking) Ada programs. HP's implementation of Ada sets up non-blocking I/O by default on interactive files if the program contains tasks. If the Ada program contains no task structures (that is, it is a sequential program), blocking I/O is set up on interactive files. You can override the defaults with binder options.

The binder option -B sets up blocking I/O and the binder option *-b* sets up non-blocking I/O. In non-blocking I/O, a task (or Ada main program) will not block when attempting interactive input if data is not available. In that case, the called HP-UX I/O routine returns immediately to the Ada run time (this is the non-blocking feature applied to Ada code as the caller). The routine's return allows the Ada run time to place the task that requested I/O on a delay queue, and to awaken the task when the I/O operation is complete. This arrangement allows other tasks to continue execution; the task requesting I/O will be delayed until the I/O operation is completed by the Ada run time (on behalf of the requesting task). However, if you set up non-blocking I/O on external called subprograms, the effect is different.

The binder (or default) options set or clear the O_NDELAY flag appropriately (see open(2) in the *HP-UX Reference*). This affects both Ada I/O and HP-UX I/O operations. For more information about Ada binder options, see the *Ada User's Manual*.

Attempting interfaced subprogram input from the preconnected file stdin can produce unexpected results if an Ada program with tasking is the caller of the external subprogram. In this situation, since the caller is a tasking program, non-blocking I/O is the default. For example, consider the case of an Ada call to an interfaced C subprogram that calls the HP-UX subroutine getchar(3). The getchar function may unexpectedly receive a (-1) result because no character may be available on stdin at the instant the call is made since non-blocking I/O is in effect.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..254 => 'A', 255 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..254 => 'A', 255 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..127 | 129..255 => 'A', 128 => '3')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..127 | 129..255 => 'A', 128 => '4')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..252 => '0', 253..255 => "298")

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..255 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..128 => 'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..126 => 'A', 127 => '1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..235 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	FILE_NAME_LONGER_THAN_14_CHARS/DIR_NOT_THERE
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	not_there/*/*/*/not_there
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'L'ST and DURATION'LAST or any value in the range of DURATION.	100_000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	not_there//not_there/*^
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	not_there/not_there/not_there/./:/not_there///
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483647
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2_147_483_648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100_000_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..252 => '0', 253..255 => "11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..251 => '0', 252..255 => "F.E:'
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 | 255 => '"', 2..254 => 'A')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483647
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FF_FF_FF_FD#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.

- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.

- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.

- C35502P: The equality operators in lines 62 and 69 should be inequality operators.

- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.